LISP SYSTEM IMPLEMENTATION

LISP SYSTEM IMPLEMENTATION

Nils M Holm

All code in this book, *but not the prose*, is provided under the Creative Commons Zero license, i.e. it can be used for any purpose, without attribution:
https://creativecommons.org/publicdomain/zero/1.0/

LISP System Implementation

Nils M Holm, 2019

Cover art by Nils M Holm

Print and Distribution:
Lulu Press, Inc.; Morrisville, NC; USA

To Life

Table of Contents

1.	Preface	6

PART I – C CODE

2.	Prelude	13
2.1	Tunable Parameters	13
3.	Data Structures	17
3.1	Special Objects	17
3.2	Memory Pools	18
3.3	Tagged Data Types	22
3.3.1	Basic Constructors and Accessors	23
3.3.2	Type Predicates	24
3.4	Abstract Machine Opcodes	27
4.	Interlude: Output Functions	29
5.	Error Handling	30
6.	Low-Level Input/Output	36
7.	Memory Management	41
7.1	Object Table Marking	42
7.2	Node Collection	47
7.2.1	The DSW Graph Marking Algorithm	48
7.3	Node Allocation	58
7.4	Vector Pool Compaction	60
7.5	Vector Allocation	65
7.6	GC Protection	66
8.	High-Level Data Types	68
9.	Some Useful List Functions	83
10.	High-Level Port I/O	86
11.	The Global Environment	90

12.	The Reader	92
13.	The Printer	115
14.	The Compiler	127
14.1	Syntax Analysis	129
14.2	Closure Conversion	137
14.2.1	Building an Environment	140
14.2.2	Finding Free Variables	144
14.2.3	Lambda Functions	148
14.2.4	Lambda Lifting	154
14.2.5	Global Definitions	162
14.2.6	Expression Conversion	163
14.3	The Abstract Machine	168
14.3.1	The Components of the LAM	170
14.3.2	The Instruction Set of the LAM	173
14.4	Code Generation	186
14.4.1	Object Table Management	186
14.4.2	Bytecode Generation	192
15.	Macro Expansion	226
16.	Built-In Functions	239
16.1	Arithmetic Functions	239
16.2	Character Functions	245
16.3	String Functions	247
16.4	Vector Functions	254
16.5	Input/Output Functions	258
16.6	List Functions	261
16.7	Type Conversion Functions	263
16.8	Program Loading	268
16.9	Heap Images	270
16.10	Miscellaneous Functions	278
17.	The Abstract Machine	282
17.1	Non-Local Exits	291
17.2	The Interpreter Loop	295
17.3	EVAL	318
18.	The REPL	323

19. Startup and Initialization 327

20. The Command Line Interface 335

PART II – LISP CODE

21. Derived Operations 345
21.1 Simple Definitions 345
21.2 Bootstrapping 347
21.2.1 Binding Constructs 352
21.3 Conditional Evaluation 356
21.4 Binding Forms 359
21.5 Higher-Order Functions 365
21.6 List Functions 368
21.7 Equivalence 371
21.8 Arithmetic Functions 375
21.9 String Functions 378
21.10 Input/Output Functions 378
21.11 Non-Local Exits 381
21.12 Image Files 386
21.13 Built-In Functions 387

PART III – ADDENDA

22. A Sample Run 401

23. Performance 406

24. Aspects of Compilation 408
24.1 Compiling LISP-N 408
24.2 Stand-Alone Compilation 410
24.3 Compiling to Machine Code 413
24.3.1 Architectural Differences 413
24.3.2 Object Table Management 414
24.3.3 Downward-Growing Stacks 415
24.3.4 Machine Registers 415
24.3.5 Generating Core Instructions 416
24.3.6 Generating Inline Functions 418
24.3.7 Code Synthesis 418

APPENDIX

25.	Using LISP9	423
25.1	Building the System	423
25.2	Interacting with LISP9	424
26.	LISP9 Summary	426
26.1	Definitions	426
26.2	Syntax	427
26.3	Binding Forms	429
26.4	Functions and Function Application	430
26.5	Conditional Evaluation	430
26.6	Loops	432
26.7	Quotation	432
26.8	List Functions	433
26.9	Mutation	435
26.10	Equivalence	436
26.11	Type Predicates	436
26.12	Type Conversion	436
26.13	Arithmetic Functions	437
26.14	Character Functions	439
26.15	String Functions	440
26.16	Vector Functions	442
26.17	Macros	443
26.18	Non-Local Exits	443
26.19	Input/Output Functions	444
26.20	System Functions	447
27.	LISP9 Compared to Scheme	450
28.	BITOP Operations	457
29.	Bibliography	458
30.	List of Figures	463
31.	Index	465

1. Preface

This book is a tour through LISP9, a LISP system with a bytecode compiler and an abstract machine for interpreting the code. LISP9 is an interactive system and takes care of a lot of things that may appear trivial from the outside, but require careful design and planning internally. It is a solid, reasonably fast, and portable version of LISP. This text describes every small detail of its implementation in great detail and without cutting corners.

The text shares a considerable intersection with *Scheme 9 from Empty Space* (S9fES), an earlier book of mine, but it concentrates on abstract machine code generation and interpretation, where S9fES employed a simple tree walker. On the other hand, S9fES also covers unbounded integer ("bignum") arithmetic and algorithms for computations on real numbers, which this book completely omits. The scope is narrower in the present work, but the depth is greater.

The LISP9 language has been inspired a lot by Scheme. R3RS Scheme was one of the first LISP systems I studied and only a short time later its successor, R4RS, appeared. The *Revised^4 Report on the Algorithmic Language Scheme* was a beautiful document and, in my opinion, the pinnacle of the development of Scheme. Later I got involved in the process that lead toward R6RS and, ironically, it is the only R^{n}RS document that lists my name in the acknowledgments. Ironically, because it was the document that made me turn my back on Scheme and, for a long time, on computing in general.

In the end, this turned out to be a good thing, because it made me realize what computer science and programming are to me. They are an expression of *love* and *beauty*. Since I spent some time at university, I have always loved to understand and explain formal systems. Formulating interconnected facts and rules in the most precise and transparent way that I can think of started to become a passion of mine. It is the reason why I write programs and why I write textbooks: to make fundamental characteristics of complex systems as obvious and clear as possible.

Unfortunately this is a lost art these days. Abelson and Sussman's epic statement [SICP1985]

Programs are meant to be read by humans
and only incidentally for computers to execute

which should be an unshakable truth for everybody who ventures to write and understand computer programs, is by many considered to be a liability today. Code must be released quickly, must be fast at any cost, must "create value" and "provide user experience". Indeed Kurlansky may have been right when he suggested that all people who fight for something will in the end adopt the traits they detest most in their enemies [Kurlansky2006]. Hacker culture has become everything that its members disdained in the May of its age: a club of entrepreneurs, marketers, business people.

Even at the risk of marginalizing myself, I refuse to be part of this new "hacker" culture. Silence, introspection, casually browsing and enjoying your own code, pride in your creation, but also, and probably most importantly, humbleness in the face of complexity are values without which the art of programming is deteriorating to craft, then trade, and then obscurity. It is no longer an art at all, but a business, and I will have no part in it!

This background is very well reflected by the LISP9 language and its implementation. The language is minimalistic and terse: a tool for the creation of art and inspiration instead of the conglomeration of massive and hyper-complex structures. Its source code attempts to be a novel written in plain language rather than an encyclopedia, and this book is its secondary literature.

The LISP9 language is so much like Scheme that many programs can probably be translated between them by substituting keywords and special objects. Its semantics should look suspiciously familiar to everybody who knows Scheme. However, its keywords are terse and it revives some traits that have their roots in more traditional LISP.

The terseness of keywords serves a pragmatic purpose: it makes the code brief and, most importantly, allows to keep lines of code

limited to a number of characters that is suitable for printing in a 6”×9” book. This is particularly helpful in the age of electronic books, because it makes program code fit comfortably in small displays.

The LISP9 implementation follows an idiomatic coding style that may need some getting used to. Here is a summary:

- function names are verbs, only sometimes nouns
- function names are short, but expressive
- global variable names are short nouns
- local variable names are single-character placeholders

At the lexical level in C code:

- CONSTANTs are all-upper-case
- Global variables are capitalized
- everything else is all-lower-case

In LISP code global variables wear `*earmuffs*`.

In the prose, keywords and global variables are typeset in **boldface** characters and local variables are typeset in *roman italics*.

Part of this style certainly runs counter to the belief that everything must be named with “obvious”, “descriptive”, and “immediately comprehensible” names. I find this explicit style utterly confusing and obtuse. Compare

```
(defun (reverse-and-concatenate
        base-list
        appended-list)
  (if (null base-list)
      appended-list
      (reverse-and-concatenate
        (cdr base-list)
        (cons (car base-list)
              appended-list))))
```

with

```
(defun (reconc a b)
  (if (null a)
      b
      (reconc (cdr a)
              (cons (car a) b))))
```

If you prefer the former style, you will have to adjust quite a bit to the style practiced in this book, but I predict that this will not be as hard as it might seem to be, and the benefit is substantial. White space is as important to code as silence is to music. Without them music is noise and code is an impenetrable wall of text.

In the concrete example, **reconc** ("reverse and concatenate") is a well-known LISP idiom, just like **strcmp** ("string-compare") is a well-known C idiom. Languages thrive on the use of idioms. They make the language brief and clear at the same time. Their only flip side is that they require some accommodation by those who are not used to them. But then you are already using terms like "loading a program" or "doing a function call" without thinking about them too much. You will get used to new idioms as easily as you got used to those.

About variables: given that you understand what **reconc** does, what will a and b be? Which one will be reversed and concatenated to which one? I would say, it is obvious from the context, and the use of "more readable" names would only make the code opaque and hard to comprehend.

This books is intended to be read front-to-back and maybe twice: once focusing on the prose and once on the code. The prose is terse, because I do not want to waste your time, so if your mind drifts for a moment, you probably *will* miss something. All concepts are introduced in a strict bottom-up order, as far as possible, so everything builds on top of previously introduced topics when going through the text in the intended order.

To avoid all issues that one or the other reader may have with the various licenses out there, the LISP9 source code is placed in the public domain or, in legislations that do not have a concept like the public domain, distributed under the Creative Commons Zero (CC0) License. This means that you, the reader, have the ultimate

freedom to do whatever you want to do with the code. So does everybody else. Freedom cannot be possessed. A copy of the CC0 can be found here:

`https://creativecommons.org/publicdomain/zero/1.0`

The complete code reproduced in this book can be found in machine-readable form on my homepage,

`http://t3x.org`

Enjoy the tour through the LISP9 system! Do not strain when things get complicated! Take a break and remember the joy of being alive!

Nils M Holm, May 2019

PART I
C CODE

2. Prelude

```
/*
 * LISP9 Interpreter
 * Nils M Holm, 2018,2019
 * In the public domain
 *
 * If your country does not have a concept like
 * the public domain, the Creative Common Zero
 * (CC0) licence applies, see
https://creativecommons.org/publicdomain/zero/1.0
 */

#define VERSION "20190411"
```

Header files are included for the following reasons: **stdlib** is needed for **exit()** return values, **stdio** for the usual buffered input and output functions, **string** and **char** for manipulating strings and characters.

The **signal** header is included, because the interpreter will intercept keyboard interrupts in order to interrupt evaluation and abort partially typed input, and **setjmp()** and **longjmp()** are used heavily in error handling.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <limits.h>
#include <signal.h>
#include <setjmp.h>
```

2.1 Tunable Parameters

The following parameters can be modified freely within some

boundaries. The **IMAGEFILE** constant specifies the file name of the heap image file that the interpreter will load at startup time, and **IMAGESRC** specifies the LISP source file to load instead of an image when no default image file exists.

```
#define IMAGEFILE    "ls9.image"
#define IMAGESRC     "ls9.ls9"
```

NNODES and **NVCELLS** indicate the number of *nodes* and *vector cells* available to the interpreter. Basically, a node is the smallest amount of storage that the interpreter can allocate. For instance, a *cons cell* consists of one single node, but most other data types, like fixnums or characters, consist of multiple nodes. A vector cell is a slot in a vector. Vectors are allocated from a separate pool.

In its default configuration, the interpreter allocates $256K$ nodes and $256K$ vector cells, which amounts to $3.25M$ bytes given an **int** size of 4 bytes. No further memory is allocated. When one of the pools runs empty, evaluation is aborted. Growing the pools dynamically would be easy, though. Use your imagination or refer to [Holm2014] for spoilers.

```
#define NNODES       262144
#define NVCELLS      262144
```

NPORTS is the maximum number of I/O ports that can be open in a LISP program at the same time. The interpreter will recycle unused port during garbage collection.

```
#define NPORTS       20
```

TOKLEN is the maximum size of all kinds of tokens, like *symbol names*, *numeric literals*, *string literals*, *character literals*, and even *comments*! The last restriction may sound weird, but it will make sense as soon as comments are explained.

```
#define TOKLEN      80
```

CHUNKSIZE is the amount of size by which all kinds of internal objects will grow, like the *symbol table*, the *object table*, the *bytecode buffer* of the compiler, and the *runtime stack* of the abstract machine.

The size of this constant is a trade-off between memory footprint and performance. Smaller values save memory, but will make the interpreter reallocate and copy structures more often. Larger values will reduce reallocation, but decrease overall memory, which will also hurt performance at some point.

The value chosen here is rather arbitrary. Feel free to conduct your own experiments!

```
#define CHUNKSIZE   1024
```

MXMAX is the maximum number of recursive calls to the *macro expander* or, in other words, the maximum number of nested macro applications. The value of 2000 may sound ridiculously large at a first glance, but it is not.

For instance, **do** expands to **labels**, **labels** expands to **let** and **let** expands to **lambda**, which in turn expands to itself, but the expander still has to perform this step. Then, each clause in **let*** expands three times: once to **let**, then to **lambda**, and then to itself.

Now keep in mind that each of these special forms can have other special forms in their bodies. And then there can be macros that expand a given number of times or traverse lists by expanding to applications of themselves. And all of these can still have all of the above in their bodies.

The value of **MXMAX** specified here is *comfortable*, but not excessive!

```
#define MXMAX       2000
```

NTRACE is the number of references to free variables that will print in case of an error. Debug information is sparse in a compiled program, so the abstract machine keeps track of the names of all free variables that are referred to during evaluation and stores them in a ring buffer. **NTRACE** is the size of that buffer.

```
#define NTRACE      10
```

PRDEPTH is the maximum depth of a structure that the printer will print before aborting. Any structure that is more than **PRDEPTH** levels "deep" (has more than **PRDEPTH** nested car fields) will be considered to be cyclic and **prin** and friends will refuse to print it.

Note that this only affects structures that cycle though **car**, but not ones that cycle through **cdr**. Limiting the latter would not make sense, because long lists are much more common than deep ones.

```
#define PRDEPTH     1024
```

3. Data Structures

All LISP data objects consist of structures in the *node pool* and *vector pool*. At C level, a **cell** is used to refer to a data object in the node pool. Vector slots are also of the type “cell”, because they also refer to objects in the pools. A cell can be thought of as a handle for a more complex object, like a fixnum, a list, or a vector. Some small (special) objects fit directly in a cell and do not allocate any space on the heap (see below). In this book, the term *heap* refers to the space allocated by the node pool and the vector pool.

Strings in LISP9 are sequences of octets in the range $0 \cdots 255$, which are represented by the data type **byte**.

Sometimes unsigned integers are needed, so **uint** is supplied here as a handy abbreviation.

```
#define cell    int
#define byte    unsigned char
#define uint    unsigned int
```

3.1 Special Objects

A *special object* is an object that does not allocate any space on the heap, but can be represented directly by a cell. All heap-based objects have positive handles, and all special objects have negative values. The **specialp()** predicate test whether an object is special.

The constants **NIL** and **TRUE** represent the LISP objects **nil** and **t**, respectively. **EOFMARK** is the object that is returned by input functions when they encounter the end of input on their associated file or device.

UNDEF is an “undefined” value that is used for different purposes, depending on its context:

- to indicate that an object is not in a hash table

- to indicate that an error was not caused by a specific object
- to indicate that a variable is not bound to a value

The **RPAREN** and **DOT** constants are used internally by the *reader* to indicate that a dot character or closing parenthesis was read, respectively.

```
#define specialp(x)  ((x) < 0)
#define NIL          (-1)
#define TRUE         (-2)
#define EOFMARK      (-3)
#define UNDEF        (-4)
#define RPAREN       (-5)
#define DOT          (-6)
```

3.2 Memory Pools

The *memory pools* hold all the LISP data objects, except for special objects, which are occasionally stored in cells. The *node pool* holds trees (graphs, actually) of *nodes* that form data objects like *fixnums*, *characters*, and *I/O ports*.

The node pool is a vector where each element has three fields: the *car field*, the *cdr field*, and the *tag field*. When the tag field is zero, the corresponding node is a *cons cell* or a *pair* and its car and cdr fields point to the elements of that pair. The car and cdr fields of a node are cells, and the tag field is a bit field that is used to store all kinds of interesting information about the node.

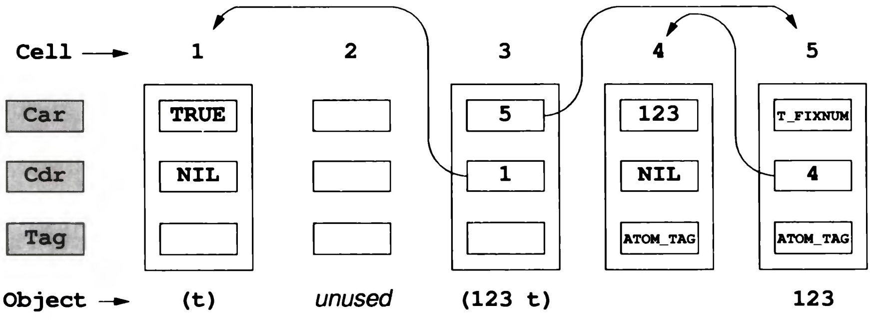


Fig. 1 – Node pool layout

More complex data objects, like fixnums, consist of multiple nodes. Figure 1 depicts a list containing the objects 123 and **T** (truth). The cons cell referring to the whole list is stored in slot 3. It is a cons cell, because its tag field does not contain any flags. Its car field points to slot 5 and its cdr field to slot 1. Slot 1 contains the special object **TRUE** in the car field and **NIL**, denoting the end of the list, in the cdr field. So node 1 alone represents the single-element list **(t)**.

Node 5, the car field or *head* of the list, has its **ATOM_TAG** flag set, indicating that it is an atomic object rather than a cons cell. The type tag **T_FIXNUM** in its car field identifies the node as the head of a fixnum. The value of the fixnum, 123, is stored in another atomic node (4) that is pointed to by the cdr field of node 5.

Note that *atomic* objects are not necessarily single-node objects. They are only atomic – indivisible – at the LISP level, but they are usually composed of multiple nodes internally.

The LISP9 node pool is composed of three arrays containing the car, cdr, and tag fields of the pool. Unsurprisingly, they are called **Car**, **Cdr**, and **Tag**. Because nodes refer to each other by index in the node pool, no pointers are necessary to implement the pool, which makes dumping and loading heap images almost trivial.

```
cell    *Car = NULL,
        *Cdr = NULL;
byte    *Tag = NULL;
```

The *vector pool* holds the values of all vector objects, such as *symbols*, *strings*, and – obviously – *vectors*. Vector objects are like atomic objects (and, in fact, they *are* atomic objects from LISP's perspective), but they have their **VECTOR_TAG** set instead of the **ATOM_TAG**. Like an atomic node, a vector node has a type tag in its car field, but its cdr field points to a cell in the vector pool instead of the node pool.

The vector pool itself is an array of cells from which consecutive sequences of cells will be allocated. It is aptly named **Vectors**.

```
cell    *Vectors = NULL;
```

The node and vector *allocators*, which will be discussed in detail later in this text, use two variables named **Freelist** and **Freevec** to keep track of free space. **Freelist** is a list of unused nodes, where the cdr field of each node points to the next free node and the last node has a cdr field of **NIL**. Allocating a node is as easy as pointing **Freelist** to its own cdr field. Similarly, **Freevec** simply points to the unallocated part of **Vectors**. A vector is allocated by adding its length to **Freevec**.

Memory allocation is cheap and simple in LISP – until the system runs out of memory. We will deal with this case later.

```
cell    Freelist = NIL;
cell    Freevec = 0;
```

Besides the atom and vector tags, there are other tag bits that can be set in a node. They will be briefly summarized here and explained later more in detail.

As explained above, both the **ATOM_TAG** and the **VECTOR_TAG** mark a node as an *atom*, i.e. as something other than a cons cell. In addition, the **VECTOR_TAG** marks it as a *vector*, i.e. an object that stores its value in the vector pool.

The **MARK_TAG**, **TRAV_TAG**, **USED_TAG**, and **LOCK_TAG** bits are used during *garbage collection*. They will be explained later in detail. Note that the **USED_TAG** and **LOCK_TAG**, although listed here, are not really tag bits, because they are kept in a different structure (see page 36).

The **PORT_TAG** marks a node as an I/O port. It is used in combination with **ATOM_TAG**, and gives a node special treatment during garbage collection.

CONST_TAG, finally, marks a node as *constant* or immutable. The LISP9 system uses this flag to indicate that the object represented by a node must not be altered by **setcar**, **vset** and similar

functions.

```
#define ATOM_TAG     0x01  /* car=type, cdr=val */
#define MARK_TAG     0x02
#define TRAV_TAG     0x04
#define VECTOR_TAG   0x08  /* car=type, cdr=val */
#define PORT_TAG     0x10
#define USED_TAG     0x20  /* port used */
#define LOCK_TAG     0x40  /* port locked  */
#define CONST_TAG    0x80
```

The **tag()**, **car()**, and **cdr()** macros are used to access the fields of a node.

```
#define tag(n)       (Tag[n])
#define car(x)       (Car[x])
#define cdr(x)       (Cdr[x])
```

The macros **caar()** through **cddddr()** are used to access elements of nested lists or pairs, but also to access the value fields of compound data objects, like fixnums. When applied to cons cells, they resemble the LISP functions of the same names. For the latter case, better names will be invented when needed.

```
#define caar(x)      (Car[Car[x]])
#define cadr(x)      (Car[Cdr[x]])
#define cdar(x)      (Cdr[Car[x]])
#define cddr(x)      (Cdr[Cdr[x]])
#define caaar(x)     (Car[Car[Car[x]]])
#define caadr(x)     (Car[Car[Cdr[x]]])
#define cadar(x)     (Car[Cdr[Car[x]]])
#define caddr(x)     (Car[Cdr[Cdr[x]]])
#define cdaar(x)     (Cdr[Car[Car[x]]])
#define cdadr(x)     (Cdr[Car[Cdr[x]]])
#define cddar(x)     (Cdr[Cdr[Car[x]]])
#define cdddr(x)     (Cdr[Cdr[Cdr[x]]])
```

```
#define caaaar(x)    (Car[Car[Car[Car[x]]]])
#define caaadr(x)    (Car[Car[Car[Cdr[x]]]])
#define caadar(x)    (Car[Car[Cdr[Car[x]]]])
#define caaddr(x)    (Car[Car[Cdr[Cdr[x]]]])
#define cadaar(x)    (Car[Cdr[Car[Car[x]]]])
#define cadadr(x)    (Car[Cdr[Car[Cdr[x]]]])
#define caddar(x)    (Car[Cdr[Cdr[Car[x]]]])
#define cadddr(x)    (Car[Cdr[Cdr[Cdr[x]]]])
#define cdaaar(x)    (Cdr[Car[Car[Car[x]]]])
#define cdaadr(x)    (Cdr[Car[Car[Cdr[x]]]])
#define cdadar(x)    (Cdr[Car[Cdr[Car[x]]]])
#define cdaddr(x)    (Cdr[Car[Cdr[Cdr[x]]]])
#define cddaar(x)    (Cdr[Cdr[Car[Car[x]]]])
#define cddadr(x)    (Cdr[Cdr[Car[Cdr[x]]]])
#define cdddar(x)    (Cdr[Cdr[Cdr[Car[x]]]])
#define cddddr(x)    (Cdr[Cdr[Cdr[Cdr[x]]]])
```

3.3 Tagged Data Types

A *tagged data object* is a data object whose internal representation consists of multiple nodes, where the root node is an atom with a *type tag* in its car field. Figure 2 illustrates the internal layout of a tagged object.

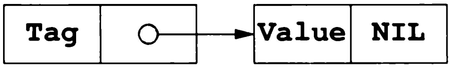


Fig. 2 – Tagged data object layout

The figure depicts the most common tagged object layout consisting of two atomic nodes. Diagrams like this one will appear frequently in this text. Each wide square box represents a node and the smaller boxes represent the car and cdr field of that node. When the box has a circle (or **NIL**) in the car field, it is a cons cell. When it has text (other than **NIL**) in the car field, it is an atom. Arrows are used to indicate references from one node to another.

Figure 2 shows an atom with a “tag” in its first node and a “value” in its other node. Like in a list, **NIL** indicates the end of the structure.

Many atoms look like this. Some have different layouts, though; they will be shown whenever such a type of atom appears in this text.

A short summary of the tagged types follows. Their details will be covered later.

T_BYTECODE marks a chunk of *bytecode* generated by the compiler. It differs from a string, because the garbage collector has to examine bytecode, but not strings. (The GC will also *recycle* strings, but their content does not matter to it.)

T_CATCHTAG can be thought of as a label for a non-local exit.

The remaining tags represent the corresponding LISP data types: characters (chars), functions (closures), small integers (fixnums), input ports, output ports, strings, symbols, and vectors.

```
#define T_BYTECODE  (-10)
#define T_CATCHTAG  (-11)
#define T_CHAR      (-12)
#define T_CLOSURE   (-13)
#define T_FIXNUM    (-14)
#define T_INPORT    (-15)
#define T_OUTPORT   (-16)
#define T_STRING    (-17)
#define T_SYMBOL    (-18)
#define T_VECTOR    (-19)
```

3.3.1 Basic Constructors and Accessors

The **cons()** and **mkatom()** macros are the principal *constructors* of the LISP9 system. A constructor is a function that allocates, initializes and returns a node (or a vector, but those will be discussed later).

Both of these macros eventually call **cons3()**, which does the actual node allocation. The only difference between them is that **cons()** creates a cons cell (like the LISP function of the same name) and **mkatom()** creates an *atomic node* – but not necessarily a complete atom!

```
#define cons(a, d)   cons3((a), (d), 0)
#define mkatom(a, d) cons3((a), (d), ATOM_TAG)
```

An *accessor* is a function that extracts the value of an atom. For atoms that have the typical structure outlined in figure 2, extracting the value is as simple as taking the **cadr** of the atom. This is, for instance, what the **portno()** macro does with I/O ports.

Things are a bit more complicated when accessing the values of *vector objects*. The details of these operations will be discussed later. In short, **string()**, **vector()**, and **symname()** extract (a pointer to) the bytes of a string, the cells of a vector, and the characters of a symbol name, respectively. **stringlen()** and **symlen()** return the lengths of the respective values in bytes and **veclen()** the length of a vector in cells. The **vecsize()** macro computes the size of a vector including its meta data. **veclink()** and **vecndx()** extract the meta data. The "link" and "index" fields indeed share the same location!

```
#define portno(n)     (cadr(n))
#define string(n)     ((byte *) &Vectors[Cdr[n]])
#define stringlen(n)  (Vectors[Cdr[n] - 1])
#define symname(n)    (string(n))
#define symlen(n)     (stringlen(n))
#define vector(n)     (&Vectors[Cdr[n]])
#define veclink(n)    (Vectors[Cdr[n] - 2])
#define vecndx(n)     veclink(n)
#define vecsize(k)    (2 + ((k) + sizeof(cell)-1)\
                       / sizeof(cell))
#define veclen(n)     (vecsize(stringlen(n)) - 2)
```

3.3.2 Type Predicates

A *type predicate* is a *predicate* (a function returning "true" or "false") that yields 1 (one), if its argument has a specific type. For instance, the **fixp()** predicate would return truth, if its argument is a fixnum and falsity for any other type of object. Note that all of the

predicates listed here return 1 in case of success and 0 in case of failure (rather than **T** and **NIL**), because they are intended for use at C level rather than LISP level.

Predicates that test for special objects can just compare their argument to the corresponding special object. For instance, **eofp()** simply compares its argument to **EOFMARK**, which represents the EOF marker internally.

All other type predicates first make sure that their arguments are *not* special objects, because in this case an invalid reference would be passed to **tag()**. They then test whether the argument object has its **ATOM_TAG** or **VECTOR_TAG** bit set (whichever is appropriate) and, if that test succeeds, check the type tag in the car field of the root node of their argument.

Each of the following macros corresponds to a LISP type.

```
#define charp(n) \
    (!specialp(n) && (tag(n) & ATOM_TAG) && \
     T_CHAR == car(n))

#define closurep(n) \
    (!specialp(n) && (tag(n) & ATOM_TAG) && \
     T_CLOSURE == car(n))

#define ctagp(n) \
    (!specialp(n) && (tag(n) & ATOM_TAG) && \
     T_CATCHTAG == car(n))

#define eofp(n) (EOFMARK == (n))

#define fixp(n) \
    (!specialp(n) && (tag(n) & ATOM_TAG) && \
     T_FIXNUM == car(n))

#define inportp(n) \
    (!specialp(n) && (tag(n) & ATOM_TAG) && \
     (tag(n) & PORT_TAG) && T_INPORT == car(n))
```

```
#define outportp(n) \
    (!specialp(n) && (tag(n) & ATOM_TAG) && \
     (tag(n) & PORT_TAG) && T_OUTPORT == car(n))

#define stringp(n) \
    (!specialp(n) && (tag(n) & VECTOR_TAG) && \
     T_STRING == car(n))

#define symbolp(n) \
    (!specialp(n) && (tag(n) & VECTOR_TAG) && \
     T_SYMBOL == car(n))

#define vectorp(n) \
    (!specialp(n) && (tag(n) & VECTOR_TAG) && \
     T_VECTOR == car(n))
```

The following are not really type predicates, except for **pairp()**, which tests whether its argument is a *cons cell*. This test is rather complex, because it has to make sure that its argument is *not* any sort of *atom*. So it complements the value of the **atomp()** predicate, which returns truth for special objects, atomic objects, and vector objects.

The *list* is a *convention* rather than a type: it is either **NIL** or a pair whose cdr part is also a list. The **listp()** predicate is a quick-and-dirty test for list objects, because it also returns truth for *dotted* or *improper* lists, i.e. lists that terminate with an atom other than **NIL**, like **(1 2 . 3)**.

The **constp()** predicate, finally, finds out whether an object has its "constant" or "immutable" bit set.

```
#define atomp(n) \
    (specialp(n) || \
     (tag(n) & ATOM_TAG) || \
     (tag(n) & VECTOR_TAG))

#define pairp(x) (!atomp(x))
```

```
#define listp(x) (NIL == (x) || pairp(x))

#define constp(n) \
    (!specialp(n) && (tag(n) & CONST_TAG))
```

3.4 Abstract Machine Opcodes

The *opcodes* (*operation codes*, *instruction codes*) of the LISP9 system will be discussed much later. However, they are needed by the garbage collector, and this is why they appear so early in this text. BTW, the LISP9 abstract machine has only 29 instructions, which are listed in the first block of the following **enum**. The remaining opcodes defined here denote inlined LISP functions.

```
enum {  OP_ILL, OP_APPLIS, OP_APPLIST, OP_APPLY,
        OP_TAILAPP, OP_QUOTE, OP_ARG, OP_REF,
        OP_PUSH, OP_PUSHTRUE, OP_PUSHVAL,
        OP_POP, OP_DROP, OP_JMP, OP_BRF, OP_BRT,
        OP_HALT, OP_CATCHSTAR, OP_THROWSTAR,
        OP_CLOSURE, OP_MKENV, OP_PROPENV,
        OP_CPREF, OP_CPARG, OP_ENTER, OP_ENTCOL,
        OP_RETURN, OP_SETARG, OP_SETREF,
        OP_MACRO,

        OP_ABS, OP_ALPHAC, OP_ATOM, OP_BITOP,
        OP_CAAR, OP_CADR, OP_CAR, OP_CDAR,
        OP_CDDR, OP_CDR, OP_CEQUAL, OP_CGRTR,
        OP_CGTEQ, OP_CHAR, OP_CHARP, OP_CHARVAL,
        OP_CLESS, OP_CLOSE_PORT, OP_CLTEQ,
        OP_CMDLINE, OP_CONC, OP_CONS, OP_CONSTP,
        OP_CTAGP, OP_DELETE, OP_DIV,
        OP_DOWNCASE, OP_DUMP_IMAGE, OP_EOFP,
        OP_EQ, OP_EQUAL, OP_ERROR, OP_ERROR2,
        OP_ERRPORT, OP_EVAL, OP_EXISTSP,
        OP_FIXP, OP_FORMAT, OP_FUNP, OP_GC,
        OP_GENSYM, OP_GRTR, OP_GTEQ, OP_INPORT,
        OP_INPORTP, OP_LESS, OP_LISTSTR,
        OP_LISTVEC, OP_LOAD, OP_LOWERC, OP_LTEQ,
```

```
OP_MAX, OP_MIN, OP_MINUS, OP_MKSTR,
OP_MKVEC, OP_MX, OP_MX1, OP_NCONC,
OP_NEGATE, OP_NRECONC, OP_NULL,
OP_NUMERIC, OP_NUMSTR, OP_OBTAB,
OP_OPEN_INFILE, OP_OPEN_OUTFILE,
OP_OUTPORT, OP_OUTPORTP, OP_PAIR,
OP_PEEKC, OP_PLUS, OP_PRIN, OP_PRINC,
OP_QUIT, OP_READ, OP_READC, OP_RECONC,
OP_REM, OP_SCONC, OP_SEQUAL, OP_SETCAR,
OP_SETCDR, OP_SET_INPORT,
OP_SET_OUTPORT, OP_SFILL, OP_SGRTR,
OP_SGTEQ, OP_SIEQUAL, OP_SIGRTR,
OP_SIGTEQ, OP_SILESS, OP_SILTEQ,
OP_SLESS, OP_SLTEQ, OP_SREF, OP_SSET,
OP_SSIZE, OP_STRINGP, OP_STRLIST,
OP_STRNUM, OP_SUBSTR, OP_SUBVEC,
OP_SYMBOL, OP_SYMBOLP, OP_SYMNAME,
OP_SYMTAB, OP_SYSCMD, OP_TIMES,
OP_UNTAG, OP_UPCASE, OP_UPPERC,
OP_VCONC, OP_VECLIST, OP_VECTORP,
OP_VFILL, OP_VREF, OP_VSET, OP_VSIZE,
OP_WHITEC, OP_WRITEC };
```

4. Interlude: Output Functions

Many functions can signal errors, and signaling an error requires some means of printing messages and data objects. Hence the most basic output functions will be defined or forward-declared here.

The **prints()** function writes a string to the currently selected output port (also called the *current output port*). Because LISP9 strings are sequences of bytes (unsigned **char**s) internally, the **printb()** macro is used to output LISP strings. It merely casts its argument to the appropriate type.

The **nl()** macro outputs a newline sequence.

The **prin()** function is the primary LISP9 output function. It is used here to print the external representations of LISP objects in error messages. It will be defined later in the "Printer" chapter.

set_outport() sets the current output port to the port number passed to it. It will be used to direct error messages to **stderr**.

```
void    prints(char *s);
void    prin(cell x);

#define printb(s)   prints((char *) s)
#define nl()        prints("\n")

int     set_outport(int port);
```

5. Error Handling

Error handling and reporting appears early in the code, because many functions must be able to signal errors.

The variables **Trace** and **Tp** form a *ring buffer* that will be used to track references to free variables in order to facilitate debugging of LISP9 code.

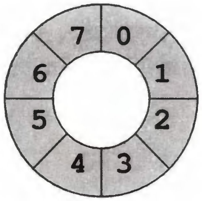


Fig. 3 – Ring buffer

The **Trace** array implements a ring of slots that will be filled cyclically while **Tp** always points to the next slot to fill. When the last slot (7) in the figure has been filled, the next element will go to slot 0 again. Because old values will be overwritten in each cycle, the buffer will always contain the **NTRACE** most recent references to free variables.

```
int     Trace[NTRACE];
int     Tp = 0;
```

The **clrtrace()** function clears the trace buffer by setting all of its slots to −1. **gottrace()** returns truth, if at least one slot of the buffer contains a reference to a variable. Variables are represented by offsets into a vector, which can never be negative.

```
void clrtrace(void) {
    int     i;

    for (i=0; i<NTRACE; i++) Trace[i] = -1;
}
```

```
int gottrace(void) {
    int     i;

    for (i=0; i<NTRACE; i++)
        if (Trace[i] != -1) return 1;
    return 0;
}
```

The **Plimit** variable sets a limit to printer output: after printing **Plimit** characters, output will be aborted. This is necessary, because errors may be caused by objects that do not have a finite representation, like cyclic lists. It also helps to make error message more digestible by only printing the first 100 characters of large objects.

Line contains the current input line number. It will only be reported when reading from a file ("loading"). When loading, the current input file name will be on top of a stack of strings that is bound to **Files**.

```
int     Plimit = 0;

int     Line = 1;
cell    Files = NIL;
```

The **report()** function reports (signals) an error by printing the message s. If the object x is not equal to **UNDEF**, it will also print the offending object x. When reading from a file, the name of the file in which the error occurred will be reported on a separate line. The file name is taken from the top of the **Files** stack.

Finally, if a trace of references to free variables exists, it will also be output on a separate line. The output procedure cycles once through **Trace**, starting at **Tp** and skipping over empty slots.

Output will be redirected to **stderr**, but at the end of **report()**, the original output port will be restored.

The declaration of **Symbols** is in fact a forward declaration. All forward declarations of variables lack an initialization part in this

text. **Symbols** is bound to a vector holding all symbol names known to the LISP9 system. It is used to report references to free variables: the slots of the **Trace** buffer point to slots of this vector.

The **ntoa()** function converts a fixnum *x* to a string representation using radix *r*, writes it to an internal buffer and returns a pointer to that buffer.

```
cell    Symbols;

char    *ntoa(int x, int r);

void report(char *s, cell x) {
    int     i, j, o;

    o = set_outport(2);
    prints("*** error: ");
    prints(s);
    if (x != UNDEF) {
        prints(": ");
        Plimit = 100;
        prin(x);
        Plimit = 0;
    }
    nl();
    if (Files != NIL) {
        prints("*** file: ");
        printb(string(car(Files)));
        prints(", line: ");
        prints(ntoa(Line, 10));
        nl();
    }
    if (gottrace()) {
        prints("*** trace:");
        i = 0;
        for (j=0; j<NTRACE; j++) {
            if (i >= NTRACE) i = 0;
            if (Trace[i] != -1) {
                prints(" ");
```

```
                    printb(
                      symname(
                        vector(Symbols)[Trace[i]]));
                }
                i++;
            }
            nl();
        }
        set_outport(o);
}
```

While the **report()** function does error reporting, the **error()** function performs *error handling*. When an *error handler* is in effect, the function passes control to that handler by *throwing* the error handler *catch tag* (or *error tag*), which is bound to the variable **Handler**. Because throwing the error tag is a non-local exit at C level, this transfer of control is implemented by performing a **longjmp()** to **Errtag**. The **Errtag** label is set up inside of the LISP9 abstract machine.

When no handler is in effect (the **Handler** variable is bound to **NIL**), the **error()** function signals an error by passing its arguments on to **report()**. After signaling the error, evaluation is aborted by passing control to the **Restart** label, which typically restarts the *REPL* (or aborts startup when used during initialization).

The **error()** function uses the **assq()**, **bindset()**, and **mkstr()** functions, which will be defined later. **assq()** searches an association list (in this case the deep-binding copy of the *global environment*), **bindset()** binds a new value to a global variable, and **mkstr()** creates a string object.

Glob, **S_errtag**, and **S_errval** are forward declarations of variables bound to the global environment, the error tag, and the *error value* (the value returned by an error handler). A special case is triggered when the error value is equal to the error tag. In this case the error value is set to s, the error message passed to **error()**.

Note that the two **jmp_buf** definitions are not forward declarations,

even if they are not being initialized.

```
jmp_buf Restart;
jmp_buf Errtag;
cell    Handler = NIL;

cell    Glob;
cell    S_errtag, S_errval;

int     assq(cell x, cell a);
void    bindset(cell v, cell a);
cell    mkstr(char *s, int k);

void error(char *s, cell x) {
    cell    n;

    n = assq(S_errtag, Glob);
    Handler = (NIL == n)? NIL: cadr(n);
    if (Handler != NIL) {
        n = assq(S_errval, Glob);
        if (n != NIL && cadr(n) == Handler)
            bindset(S_errval,
                    mkstr(s, strlen(s)));
        longjmp(Errtag, 1);
    }
    report(s, x);
    longjmp(Restart, 1);
}
```

The **expect()** function generates an error of the form

who: **`expected`** *what*: *got*

where *who* and *what* are replaced by the corresponding parameters: the name of the calling function and the expected type. The *got* parameter it a reference to the object that was received in the place of an object of the desired type. This function is a short cut for reporting type errors.

```
void expect(char *who, char *what, cell got) {
    char    b[100];

    sprintf(b, "%s: expected %s", who, what);
    error(b, got);
}
```

The **fatal()** function is called whenever the LISP system ends up in a condition where no recovery is possible. For instance:

- being unable to allocate the memory pools
- finding no open input or output port
- attempting to load a corrupted image file
- encountering an error during compilation of the initial heap image

Calling **fatal()** will print a message and terminate the LISP9 process.

```
void fatal(char *s) {
    fprintf(stderr, "*** fatal error: ");
    fprintf(stderr, "%s\n", s);
    exit(EXIT_FAILURE);
}
```

6. Low-Level Input/Output

All LISP9 *input/output* operations are performed on *I/O ports* (or just *ports*). The following data structures and functions implement I/O ports.

Each port object contains a value called the *port number*. Internally, this number is an offset into the **Ports** and **Port_flags** arrays. The **Ports** array basically maps port objects to C **FILE**s and **Port_flags** contains the garbage collector bits used in combination with ports.

When the system starts up, it will create three ports and associate them with the **stdin**, **stderr**, and **stdout** **FILE**s. Their port numbers are stored in the variables **Inport**, **Outport**, and **Errport**.

```
FILE    *Ports[NPORTS];
char    Port_flags[NPORTS];

int     Inport = 0,
        Outport = 1,
        Errport = 2;
```

The I/O functions can also read and write strings instead of ports. When binding **Outstr** to a string of the length **Outmax** and setting **Outptr** to zero, all output will go to that string instead of the current output port. The string will grow automatically to create space for any characters sent to it. When output is complete, the actual size of **Outstr** can be found in **Outptr**. The output should then be copied to a string of the proper size.

Reading from a string works similarly. When pointing **Instr** to a C string and setting **Rejected** to -1, all subsequent input will be read from that string instead of the current input port. When the end of the string is reached, reading another character from it will yield **EOF**.

```
cell    Outstr = NIL;
```

```
int     Outmax = 0;
int     Outptr = 0;

char    *Instr = NULL;
char    Rejected = -1;
```

The **readc()** function is the principal input function of the system. *All* input will pass through this interface. Whenever **readc()** is called, it will return the next character from the current input port (or a string, when **Instr** is not **NULL**). When the associated file, device, or string cannot deliver any further input, the function will return **EOF**.

A fatal error will be signalled, if the current input port is not open when calling **readc()**.

```
int readc(void) {
    int     c;

    if (Instr != NULL) {
        if (Rejected > -1) {
            c = Rejected;
            Rejected = -1;
            return c;
        }
        if (0 == *Instr) {
            return EOF;
        }
        else {
            return *Instr++;
        }
    }
    else {
        if (NULL == Ports[Inport])
            fatal("readc: input port "
                  "is not open");
        return getc(Ports[Inport]);
    }
```

```
}
```

The **rejectc()** function puts the character *c* back to the current input source, which may be a string or a port. When the source is read for the next time, it will produce that character again.

```
void rejectc(int c) {
    if (Instr != NULL) {
        Rejected = c;
    }
    else {
        ungetc(c, Ports[Inport]);
    }
}
```

The **flush()** function writes all pending output to the file or device associated with the current output port. In case of an error it uses the **mkport()** function (which creates a port object) to wrap the current output port number up in a port object. This is necessary because the **error()** function expects a LISP object as its second argument and not a C integer.

```
cell    mkport(int p, cell t);

void flush(void) {
    if (fflush(Ports[Outport]))
        error("file write error, port",
            mkport(Outport, T_OUTPORT));
}
```

The **blockwrite()** function is the principal output function of the LISP9 system. All output passes through this interface. The function writes *k* characters from the string *s* to the current output port – or to an output string, if **Outstr** is not **NIL**.

When writing to the **stdout** (1) or **stderr** (2) port *and* the last character in the output is a newline character, **blockwrite()** will

flush the current output port after writing.

blockwrite() will also subtract *k* from the **Plimit** variable and not write any further output when its value goes below 1. Note that **Plimit** will be ignored, if it is 0 initially.

A fatal error will be signalled, if the current output port is not open when calling **blockwrite()**.

```
void blockwrite(char *s, int k) {
    cell    n;

    if (1 == Plimit) return;
    if (Outstr != NIL) {
        while (Outptr + k >= Outmax) {
            n = mkstr(NULL, Outmax+1000);
            memcpy(string(n), string(Outstr),
                    Outptr);
            Outmax += 1000;
            Outstr = n;
        }
        memcpy(&string(Outstr)[Outptr], s, k);
        Outptr += k;
        string(Outstr)[Outptr] = 0;
        return;
    }
    if (NULL == Ports[Outport])
        fatal("blockwrite: output port "
              "is not open");
    if (fwrite(s, 1, k, Ports[Outport]) != k)
        error("file write error, port",
            mkport(Outport, T_OUTPORT));
    if ((1 == Outport || 2 == Outport) &&
        '\n' == s[k-1]
    )
        flush();
    if (Plimit) {
        Plimit -= k;
        if (Plimit < 1) Plimit = 1;
```

```
    }
}
```

The **writec()** function writes the single character *c* to the current output port (or string), and **prints()** writes the string *s*.

```
void writec(int c) {
    char    b[1];

    b[0] = c;
    blockwrite(b, 1);
}

void prints(char *s) {
    blockwrite(s, strlen(s));
}
```

7. Memory Management

The core of every LISP memory management system is the *garbage collector*, often abbreviated *GC*. All LISP data objects have *indefinite extent*, which means that once created, they will continue to exist till the end of the world (which, in this particular case, happens when the LISP system shuts down). Because LISP never deallocates memory, though, even simple programs may run out of memory soon. This is where the garbage collector comes into play. As soon as a LISP object cannot be accessed any longer by any LISP program in memory, it does not make any difference whether it exists or not. Deleting it would not violate the principle of indefinite extent, because the system has already "forgotten" it.

A historical note: the early models of the Symbolics 3600 LISP Machine did not have a garbage collector [HOPL1993]. Because LISP Machine programs typically avoided excessive consing, programs would often run for hours or even days before memory started to run low. The user would when "save the world" (dump a heap image to disk), restart the machine and reload the image, which reconstructed the free lists and hence effectively performed a garbage collection. A garbage collection could be initiated manually, but it could take such a long time that the save, restart, load cycle was often faster. Later models of the 3600 provided an ephemeral (generational) collector [Baker1978] [Moon1984], which performed garbage collection concurrently while the CPU was interpreting LISP code.

Compared to the ephemeral GC by Baker and Moon, the LISP9 memory management system is rather simple. It consists of two parts: a mark-and-sweep collector that recycles nodes in the node pool and a compacting collector that defragments the vector pool. In addition to reclaiming memory, the LISP9 collector also finalizes unaccessible I/O ports and maintains the object table, which contains all data objects that are contained as literals in LISP programs.

The first step in memory management is, of course, to allocate the memory that is to be managed by the garbage collector. The

LISP9 system allocates the three arrays **Car**, **Cdr**, and **Tag** that comprise the node pool, as well as the single array **Vectors** that will be used to store the values of vector objects.

Pool allocation is done in the two functions **alloc_nodepool()** and **alloc_vecpool()**. Failing to allocate sufficient memory will signal a fatal error.

```
void alloc_nodepool(void) {
    Car = malloc(sizeof(cell) * NNODES);
    Cdr = malloc(sizeof(cell) * NNODES);
    Tag = malloc(NNODES);
    if (NULL == Car ||
        NULL == Cdr ||
        NULL == Tag
    )
        fatal("alloc_nodepool: "
              "out of physical memory");
    memset(Car, 0, sizeof(cell) * NNODES);
    memset(Cdr, 0, sizeof(cell) * NNODES);
    memset(Tag, 0, NNODES);
}

void alloc_vecpool(void) {
    Vectors = malloc(sizeof(cell) * NVCELLS);
    if (NULL == Vectors)
        fatal("alloc_vecpool: "
              "out of physical memory");
    memset(Vectors, 0, sizeof(cell) * NVCELLS);
}
```

7.1 Object Table Marking

The discussion of memory management starts, maybe, with the most unobvious aspect of garbage collection: the reclamation of unused *data objects*. Most LISP programs will contain lots of literal data objects, like strings, numbers, and (quoted) lists. For instance, even the little program

```
(defun (f x)
  (if (= x 0)
      nil
      (cons 1 (f (- x 1)))))
```

contains the data objects **0**, **nil**, and **1**. As long as the program is stored in memory as an *abstract syntax tree* (*AST*) in the form of a list structure that is bound to some (LISP) variable, the data objects will never be reclaimed. However, once the program is compiled to abstract machine code, the AST is no longer needed and can be recycled. If no measures are taken to protect the data objects, they will be recycled together with the rest of the AST.

This is why LISP9 maintains an *object table* (*OBTAB*) that contains all data objects that are referenced in compiled code. That table is implemented as a LISP vector and bound to the variable **Obarray** at C level.

Many objects in the table are ephemeral, though. For example, the form

```
(def foo '(1 2 3))
```

binds the data object **(1 2 3)** to the variable *foo*. After binding the object to a variable, it does not have to be kept in the object table any longer, because the binding to the variable protects it from being reclaimed during GC.

Therefore, the LISP9 collector *marks* all objects in the OBTAB that are still referenced by some code as "used", so that the collector can later recycle all objects that are not marked as "used". This approach is called *mark and sweep* garbage collection. It is a two-phase method where the first phase marks all "live" (i.e. accessible) objects and the second phase recycles "dead" (inaccessible) ones and makes them available for future allocation.

The collector described here marks OBTAB slots in a separate byte vector that is bound to **Obmap**. When space is a critical constraint, this map may be implemented as a bit vector (using a slightly different algorithm), but a byte vector with three states per byte will be used here for the sake of simplicity.

Each byte in **Obmap** corresponds to one slot in **Obarray.** When

marking starts, all bytes are either "free" or "allocated" or "used", indicated by the values **OBFREE**, **OBALLOC**, and **OBUSED**, as defined below. The collector will then change all **Obmap** slots whose corresponding **Obarray** slots that can be accessed through a compiled program to **OBUSED**. At the end of the collection, all slots that are not "used" will be changed to "free", so a different object can be stored in them.

```
#define OBFREE     0
#define OBALLOC    1
#define OBUSED     2
```

One major complication of the approach outlined here is that the mark procedure processing the object table has to "know" enough about the abstract machine code of the system to be able to locate *quote instructions*, which load objects from the object table into a register of the abstract machine. This is complicated, because object references (quote instructions) can appear anywhere in a compiled program, as illustrated in figure 4.

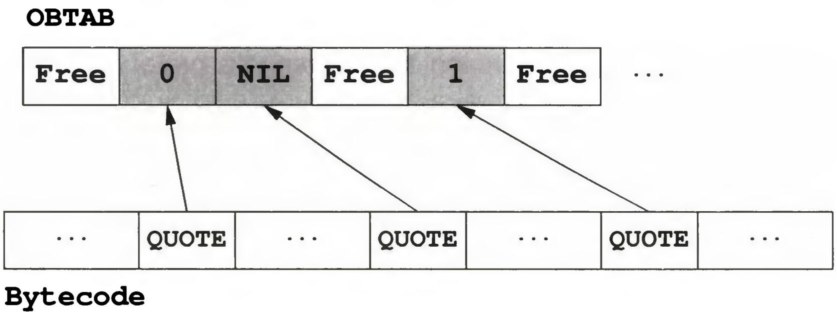


Fig. 4 – Object table

In the system described here, where the abstract machine is designed carefully in order to deal with such details, this is not a big problem. Even when generating code for RISC architectures, where each instruction has the same length, such an approach is probably feasible. However, when dealing with architectures where instruction sizes vary wildly within broad boundaries, this method quickly becomes impractical.

In such scenarios, a different mechanism has to be used. For instance, the compiler could extract all literal data objects from a program, store them in a list, and generate a single instruction at the beginning of each chunk of machine code that skips over the address of the list. In this case the garbage collector only has to know the offset of the list address inside of the chunk of code.

In the present implementation, though, the mark procedure will traverse abstract machine programs, locate quote instructions, and mark slots in the OBTAB accordingly.

The LISP9 abstract machine code has zero-operand instructions with a size of one byte, single-operand instructions with a size of three bytes, and two-operand instructions with as size of five bytes. Their sizes are defined by the **ISIZE0**, **ISIZE1**, and **ISIZE2** constants below.

```
#define ISIZE0      1
#define ISIZE1      3
#define ISIZE2      5
```

Arguments are stored in a program byte vector using big-endian byte ordering. The **fetcharg()** macro is used to retrieve an argument at position i from byte vector a.

```
#define fetcharg(a, i) \
    (((a)[i] << 8) | (a)[(i)+1])
```

The **marklit()** function marks literal data objects referenced in program *chunk* (byte vector) p. p is bound to a byte vector containing abstract machine instructions. All the function does is to walk through the code, identify instructions with arguments, and skip over their arguments. When it finds a quote instruction (**OP_QUOTE**), it also fetches its argument and marks the corresponding OBTAB slot as “used”.

```
cell    Obarray, Obmap;
```

```
void marklit(cell p) {
    int     i, k, op;
    byte    *v, *m;

    k = stringlen(p);
    v = string(p);
    m = string(Obmap);
    for (i=0; i<k; ) {
        op = v[i];
        if (OP_QUOTE == op) {
            m[fetcharg(v, i+1)] = OBUSED;
            i += ISIZE1;
        }
        else if (OP_ARG == op ||
                 OP_PUSHVAL == op ||
                 OP_JMP == op ||
                 OP_BRF == op ||
                 OP_BRT == op ||
                 OP_CLOSURE == op ||
                 OP_MKENV == op ||
                 OP_ENTER == op ||
                 OP_ENTCOL == op ||
                 OP_SETARG == op ||
                 OP_SETREF == op ||
                 OP_MACRO == op)
        {
            i += ISIZE1;
        }
        else if (OP_REF == op ||
                 OP_CPARG == op ||
                 OP_CPREF == op)
        {
            i += ISIZE2;
        }
        else {
            i += ISIZE0;
        }
    }
```

```
}
```

7.2 Node Collection

The *node collector* is the central part of the LISP9 garbage collector. All objects in the heap have their roots in the node pool: fixnums, chars, and ports consist of two nodes that are linked together in the node pool. Strings, symbols, and vectors are nodes that link to some sequence of cells in the vector pool. Lists, trees, and graphs are collections of nodes in the node pool. No object without a root in the node pool can ever be visible to the interpreter.

The *GC roots* (garbage collector roots) form a set of cells that reference root nodes in the node pool. GC root nodes must never be recycled by the node collector. Such nodes include, for example:

- the symbol table
- the object table
- the runtime stack
- the global environment
- the registers of the abstract machine

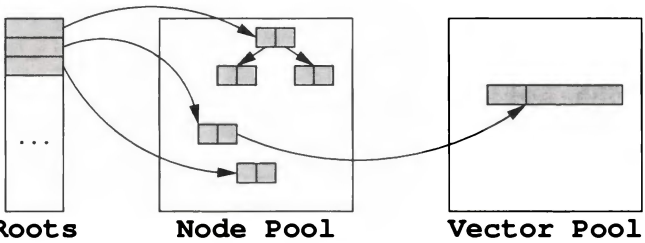


Fig. 5 – Live nodes in the heap structure

Figure 5 depicts the heap structure with root cells pointing to nodes in the node pool, which may in turn point to other nodes or to vectors in the vector pool. In fact, vector pool elements may also point back to the node pool. We will get back to this detail.

The task of the *garbage collector* is to find nodes that are "live" (i.e. pointed to by a GC root), mark them, and then "sweep" the

heap by adding unmarked nodes to the free list. It will also take care of vectors, but this is done in a different part of the collector.

The hardest part of mark and sweep GC is to mark the live graphs in the node pool. The task is superficially easy to solve as follows:

(1) For each atom in the set of GC roots R, mark the atom and then mark the cdr part of the atom.

(2) For each cons cell in R, mark the cell itself and then mark the car part and the cdr part of the cell.

(3) For each vector in R, mark the vector node and then mark each element of the vector in the vector pool (where each element points back to the node pool).

And then there is the trivial case:

(0) For each special object and each already-marked object in R, do nothing.

The algorithm is straightforward enough, but there is one problem: it may recurse on both the car and cdr field of nodes, which means that the stack space required by it equals the length of the longest path of any disconnected graph in the node pool. For instance, marking a list of $10,000$ elements would recurse $9,999$ times.

This is why the node collector uses an extended version of the *Deutsch-Schorr-Waite (DSW) graph marking algorithm* for traversing the GC roots. The DSW algorithm, which is described in [SW1967] and which was independently discovered by L. Peter Deutsch [Knuth1968], marks any graph in constant space and requires only one single cell of additional storage.

The implementation presented in the following also marks all n-ary trees (vectors) in constant space using one cell of additional space per vector.

7.2.1 The DSW Graph Marking Algorithm

The problem solved by the DSW algorithm is a follows: given a tree with root node N (as shown in figure 6), set all M bits in the tree to one using only one additional cell of storage ($Parent$).

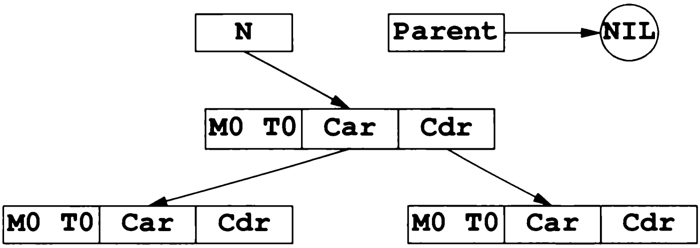


Fig. 6 – GC state M0,T0 (unmarked)

When the algorithm descends into a car or cdr subtree, it stores the reference to the current node in *Parent*. However, *Parent* may already be used for another (grandparent) reference, so it first stores *Parent* in the car/cdr field that leads to the subtree to traverse. When the subtree has been marked, it restores N from *Parent* and *Parent* from the car/cdr field. This way, it uses the tree itself as a stack for storing the references of nodes to return to.

In the following, the individual states of node traversal will be explained. When traversal of a tree starts at its root, all nodes of the tree are in state $M0, T0$, meaning that they have both their **MARK_TAG** and their **TRAV_TAG** cleared.

The first thing the algorithm does is to set both the M (**MARK_TAG**) and T (**TRAV_TAG**) bits of the node N. The M bit marks the node as "live" and the T bit indicates that traversal of the node is not yet finished (because its cdr subtree is still unmarked). The node must be marked immediately in order to avoid infinite traversal in cyclic graphs.

The next step is to visit the car node, which is done by moving

- $X = car(N)$
- $car(N) = Parent$
- $Parent = N$
- $N = X$

where X is an auxiliary register. (Note: X is not an additional location, it just makes parallel assignment explicit.) This step leaves the tree in the state depicted in figure 7, where the original root node, now being pointed to by *Parent*, is in state $M1, T1$.

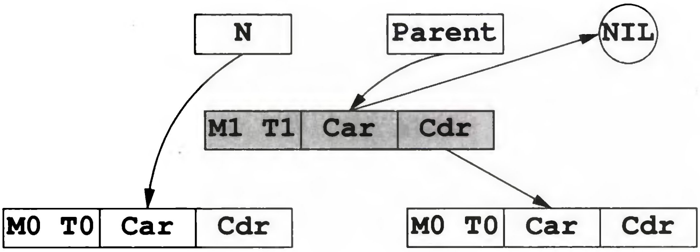


Fig. 7 – GC state M1,T1 (marked, car traversal in progress)

When the car part of the tree has been completely marked (by applying the DSW algorithm recursively), the process returns to *Parent*. Traversal of the car part is now complete, so the T bit of *Parent* is cleared. In order to traverse the cdr part of the node, the following assignments will be performed in the tree in figure 7:

- $X = cdr(Parent)$
- $cdr(Parent) = car(Parent)$
- $car(Parent) = N$
- $N = X$

These modifications leave the root node in state $M1, T0$, as shown in figure 8. It moves the grandparent reference from the car field of the root node to its cdr field and points N to the cdr part.

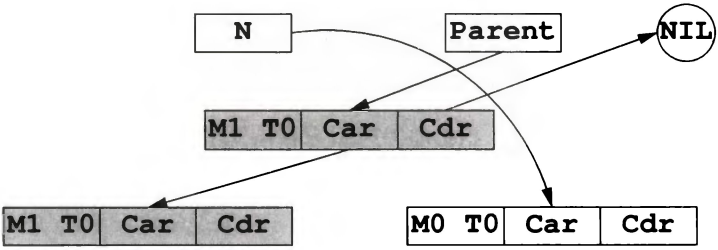


Fig. 8 – GC state M1,T0 (marked, car traversal complete)

While the root node of the tree is now in "completed" state, N still points to the cdr part of the root node, so that subtree will also be marked, again by applying the DSW algorithm recursively. When marking of the cdr part completes, the algorithm returns to *Parent* again, this time finding it in state $M1, T0$ (complete). It then restores the original tree from the configuration in figure 8:

- $X = cdr(Parent)$
- $cdr(Parent) = N$
- $N = Parent$
- $Parent = X$

The tree is now in its final state with all links restored to their original destinations and all M bits set in the entire tree (figure 9).

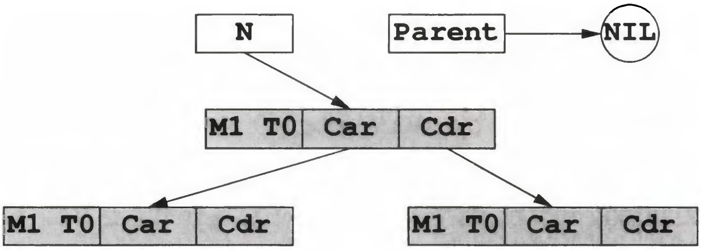


Fig. 9 – GC state M1,T0 (marked, final)

The DSW graph marking algorithm is also called a *pointer reversal* algorithm, because it reverse pointers that normally point "forward" to their child nodes so that they point "backwards" to their parents. See figure 10, which shows a tree in the middle of traversal.

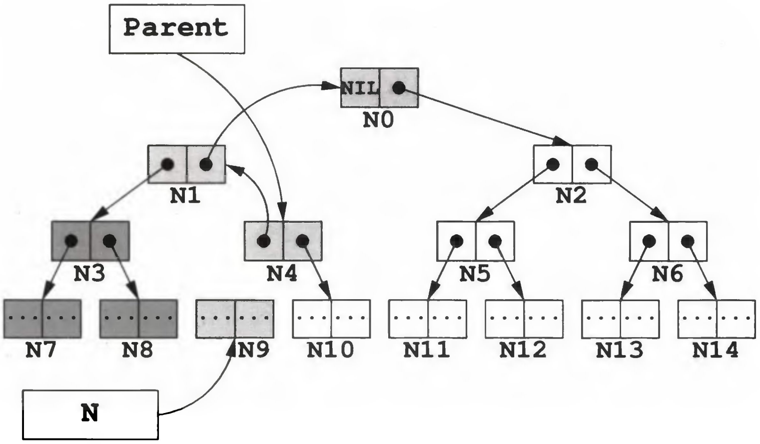


Fig. 10 – Pointer reversal in the DSW algorithm

N_9, which is the currently visited node, stores its parent in *Parent*, N_4 stores its parents in its car field, N_1 stores its parent in its cdr field, and N_0, being the root of the tree, has no parent and hence stores **NIL** (the original value of *Parent*) in its car field. The only

additional storage cell is needed to hold the parent of the node currently being visited.

In the figure, the nodes N_3, N_7, and N_8 are completely marked, N_0, N_1, and N_4 are partially marked and will be returned to, N_9 is currently in the process of being marked, and the other nodes have not yet been visited at all.

The algorithm finishes when it returns from a node and finds that *Parent* equals **NIL**.

In addition to graph marking as described above, the **mark()** function, which implements the mark stage of the garbage collector, also marks vectors, bytecode, and I/O ports.

In order to identify tagged data types during garbage collection, the function matches the currently visited node against a tree of the shape displayed in figure 2 on page 22, i.e. an atom node whose cdr field is not **NIL** and whose car field holds the desired *type tag*. If the cdr field was allowed to be **NIL**, the current node could be a value field of a fixnum atom, which would then be erroneously matched, if its value was equal to the value of the desired type tag.

Once an atom is found, it is processed as follows:

For *port objects*, the collector will set the **USED_TAG** bit of the corresponding port and for *bytecode* chunks, it will delegate marking to the **marklit()** function discussed earlier in this chapter.

Marking vectors is a generalization of marking cons cells. Instead of using the **TRAV_TAG** bit to direct traversal, though, an "index" field in the meta data of the vector is used to keep track of traversal.

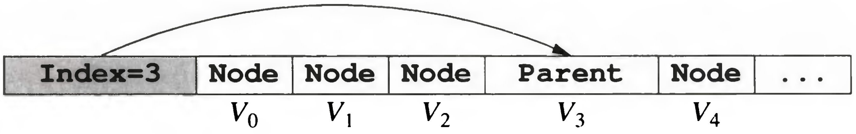


Fig. 11 – Use of the index field in vector traversal

When traversal of a vector V starts, the index field is set to 0, and *Parent* is stored in V_0, the first slot of the vector. When the node stored in the first slot has been marked, the index is incremented

and the process repeats until the index reaches the length of the vector. Figure 11 shows a vector V while traversing V_3.

Marking a vector also involves setting its "link" field to the node that refers to the vector. This is not important at the moment, we will get back to this detail later in this chapter.

The following **mark()** function marks all nodes, vector objects, ports, and literal objects that are accessible through the argument n.

```
/*
 * Traversal states:
 * S0 = M0,T0
 * S1 = M1,T1
 * S2 = M1,T0
 */

void mark(cell n) {
    cell    x, parent, *v;
    int     i;

    parent = NIL;
    while (1) {
        if (specialp(n) || (tag(n) & MARK_TAG)) {
            if (NIL == parent)
                break;
            /* S1 --> S1|done */
            if (tag(parent) & VECTOR_TAG) {
                i = vecndx(parent);
                v = vector(parent);
                if (tag(parent) & TRAV_TAG &&
                    i+1 < veclen(parent)
                ) {
                    /* S1 --> S1 */
                    x = v[i+1];
                    v[i+1] = v[i];
                    v[i] = n;
                    n = x;
```

```
                vecndx(parent) = i+1;
            }
            else {
                /* S1 --> done */
                x = parent;
                parent = v[i];
                v[i] = n;
                n = x;
                veclink(n) = n;
            }
        }
        /* S1 --> S2 */
        else if (tag(parent) & TRAV_TAG) {
            x = cdr(parent);
            cdr(parent) = car(parent);
            car(parent) = n;
            tag(parent) &= ~TRAV_TAG;
            n = x;
        }
        /* S2 --> done */
        else {
            x = parent;
            parent = cdr(x);
            cdr(x) = n;
            n = x;
        }
    }
    /* S0 --> S1 */
    else if (tag(n) & VECTOR_TAG) {
        tag(n) |= MARK_TAG;
        if (T_VECTOR == car(n) &&
            veclen(n) != 0)
        {
            tag(n) |= TRAV_TAG;
            vecndx(n) = 0;
            v = vector(n);
            x = v[0];
            v[0] = parent;
```

```
                parent = n;
                n = x;
            }
            else {
                veclink(n) = n;
            }
        }
        /* S0 --> S2 */
        else if (tag(n) & ATOM_TAG) {
            if (cdr(n) != NIL) {
                if (T_BYTECODE == car(n))
                    marklit(cdr(n));
                else if (T_INPORT == car(n) ||
                         T_OUTPORT == car(n)
                )
                    Port_flags[portno(n)]
                                  |= USED_TAG;
            }
            x = cdr(n);
            cdr(n) = parent;
            parent = n;
            n = x;
            tag(parent) |= MARK_TAG;
        }
        /* S0 --> S1 */
        else {
            x = car(n);
            car(n) = parent;
            tag(n) |= MARK_TAG;
            parent = n;
            n = x;
            tag(parent) |= TRAV_TAG;
        }
    }
}
```

The **gc()** function recycles all unaccessible objects from the node pool and places them in the freelist. It will not touch the vector

pool, which will be garbage-collected by a different function.

Garbage-collection of the node pool involves the following steps:

- All ports that have their **LOCK_TAG** bit set will be marked by setting their **USED_TAG** bit, all others will be unmarked
- All objects that can be accessed through a member of the **GC_roots** array will be marked
- All unmarked nodes are consed to **Freelist** and all other nodes have their **MARK_TAG** bit cleared
- All ports that do not have their **USED_TAG** bit set will be closed and their **Ports** slots will be freed
- All *OBTAB* slots that are not marked **OBUSED** will be cleared

The **GC_roots** array, the runtime stack **Rts**, and the stack pointer **Sp** will be defined later. When the **GC_verbose** flag is set, it will cause some information to print during garbage collection. This option is intended for debugging the LISP system.

Note that the runtime stack **Rts** gets some special treatment during garbage collection. The stack is a vector of some fixed size (which may grow during evaluation), but only elements in its first $Sp+1$ slots are actually live objects, because objects with slot indexes above the stack pointer are no longer accessible. See figure 12.

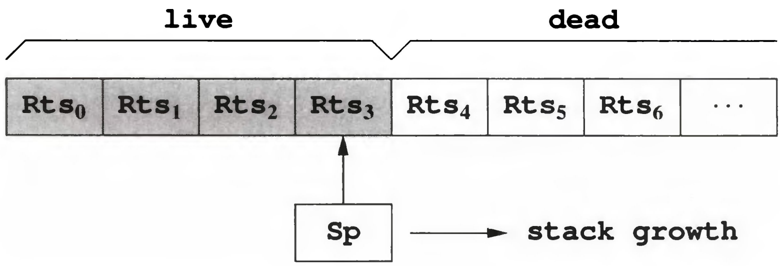


Fig. 12 – Live objects on the runtime stack

This is why the length of **Rts** is changed to $1+Sp \times sizeof(cell)$ before marking the GC roots. When the mark phase completes, the stack size is reset to the full length of the stack, so that a subsequent vector pool compaction will also copy the unused slots.

```
int       GC_verbose = 0;
cell      *GC_roots[];
cell      Rts;
int       Sp;

int gc(void) {
    int       i, n, k, sk;
    char      buf[100];
    cell      *a;
    byte      *m;

    for (i=0; i<NPORTS; i++) {
        if (Port_flags[i] & LOCK_TAG)
            Port_flags[i] |= USED_TAG;
        else if (i == Inport || i == Outport)
            Port_flags[i] |= USED_TAG;
        else
            Port_flags[i] &= ~USED_TAG;
    }
    if (Rts != NIL) {
        sk = stringlen(Rts);
        stringlen(Rts) = (1 + Sp) * sizeof(cell);
    }
    for (i=0; GC_roots[i] != NULL; i++) {
        mark(*GC_roots[i]);
    }
    if (Rts != NIL) {
        stringlen(Rts) = sk;
    }
    k = 0;
    Freelist = NIL;
    for (i=0; i<NNODES; i++) {
        if (!(tag(i) & MARK_TAG)) {
            cdr(i) = Freelist;
            Freelist = i;
            k++;
        }
```

```
        else {
            tag(i) &= ~MARK_TAG;
        }
    }
    for (i=0; i<NPORTS; i++) {
        if (!(Port_flags[i] & USED_TAG) &&
            Ports[i] != NULL)
        {
            fclose(Ports[i]);
            Ports[i] = NULL;
        }
    }
    n = veclen(Obarray);
    a = vector(Obarray);
    m = string(Obmap);
    for (i=0; i<n; i++) {
        if (OBUSED  == m[i]) {
            m[i] = OBALLOC;
        }
        else {
            m[i] = OBFREE;
            a[i] = NIL;
        }
    }
    if (GC_verbose) {
       sprintf(buf, "GC: %d nodes reclaimed", k);
       prints(buf); nl();
       flush();
    }
    return k;
}
```

7.3 Node Allocation

With the heavy lifting done by the garbage collector out of the way, node allocation is very straightforward.

In the simplest and most common case, the **cons3()** function just allocates $n = car$(**Freelist**) and sets **Freelist** to cdr(**Freelist**). See

figure 13.

It then initializes the fields of the freshly allocated node with the *pcar*, *pcdr*, and *ptag* values and returns it. This path through the function is very fast, so node allocation is cheap.

When the freelist is empty (**NIL**), a node collection is initiated. When the freelist is still empty after the collection, the evaluation in progress is aborted by signaling an error.

Both the *pcar* and *pcdr* values are GC-protected during garbage collection, i.e. they cannot be reclaimed before they are used, even if the caller did not protect them explicitly. This allows to write, for instance,

```
cons(a,b)
```

without having to protect *a* and *b* first. The GC roots **Tmp_car** and **Tmp_cdr** are used to protect the respective fields.

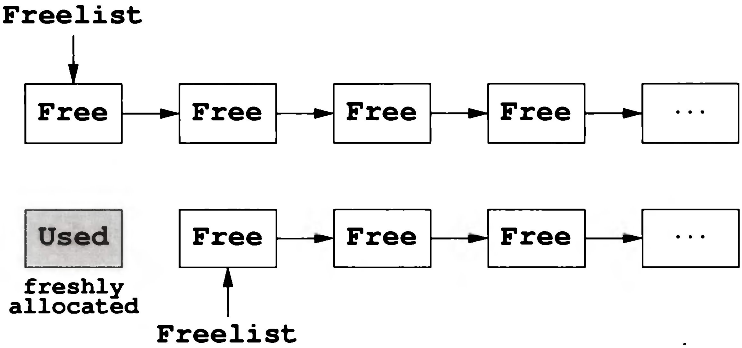


Fig. 13 – Node allocation

Note that the car fields of *atoms* and the cdr fields of *vector objects* are not node references, so they will not be protected.

When the number of free nodes falls below half the pool size after node collection, *GC thrashing* will start. That is, the garbage collector will be called very frequently and eventually the system will spend most of its time collecting nodes. This is the point where the node pool should grow, but the implementation presented here will not deal with this aspect. See [Holm2014] for an implementation that handles GC thrashing by growing the memory pools.

```
cell    Tmp_car = NIL,
        Tmp_cdr = NIL;

cell cons3(cell pcar, cell pcdr, int ptag) {
    cell    n;
    int     k;

    if (NIL == Freelist) {
        if (0 == ptag)
            Tmp_car = pcar;
        if (!(ptag & VECTOR_TAG))
            Tmp_cdr = pcdr;
        k = gc();
        if (k < NNODES / 2) {
            /* memory low! */
        }
        Tmp_car = Tmp_cdr = NIL;
        if (NIL == Freelist)
            error("cons3: out of nodes", UNDEF);
    }
    n = Freelist;
    Freelist = cdr(Freelist);
    car(n) = pcar;
    cdr(n) = pcdr;
    tag(n) = ptag;
    return n;
}
```

7.4 Vector Pool Compaction

A node that has been reclaimed can simply be reallocated because, as far as memory management is concerned, is has only two states: “used” or “free”, “live” or “dead”. Things are not that simple with vector objects, because vectors also have a *size*. So an allocator not only has to find a free vector for reallocation, it has to find a vector of the right size. When the vector is too small, it cannot be allocated and when it is too large, it will be split in two,

causing fragmentation of the vector pool.

Fragmentation of the pool (figure 14) can cause vector allocation to fail, even if sufficient memory is present in the pool, because none of the free chunks of cells is large enough to satisfy the allocation request.

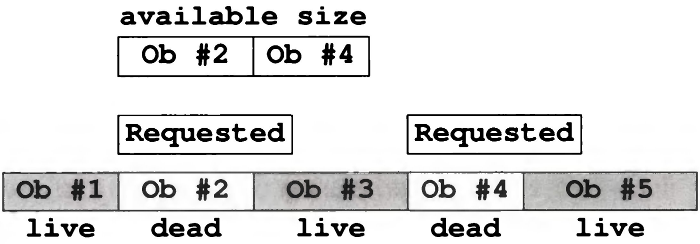


Fig. 14 – Vector pool fragmentation

There are lots of clever techniques for allocating chunks of the most appropriate size from memory pools. Pool *compaction* is a simple method that gives perfect results regarding memory utilization and has linear run time. It does involve some copying, though. During compaction all live objects move to one end of the pool and then the free space pointer is reset to the end of the live data. Because the free space is one continuous free vector after compaction, all allocation requests will succeed immediately until the pool runs full.

The *vector pool compactor* makes heavy use of the meta data of objects in the vector pool. Therefore, now is a good time to introduce the internal structure of a vector.

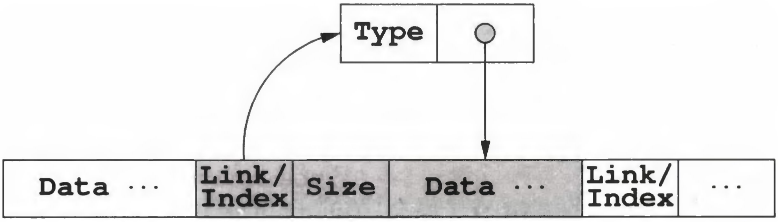


Fig. 15 – Vector pool structure

Figure 15 shows the data of a vector object in the vector pool and a vector node whose cdr field points to the data in the pool. The vector data associated with the node is shown in grey. The data of

the vector consists of four fields:

- The "link" field, which points back to the node of the vector object
- The "index" field, which is used by the node collector during graph marking
- The "size" field, which holds the size of the vector *in bytes*
- The "data" field, which contains the value of the vector object itself, like the characters of a string or the slots of a LISP vector object

Because the "link" field is only used during compaction and the "index" field is only used during marking, they share the same location in the meta data of a vector.

The offsets of the vector fields are defined by the constants **RAW_VECLINK**, **RAW_VECSIZE**, and **RAW_VECDATA**. They are called "raw" offsets, because they are used internally, where a vector starts at the "link" field. Vector *nodes* refer to the "data" field, though, so from their perspective the "size" field is at position -1 and the "link/index" field at position -2.

Note that the vector pool really is a vector of *cells* rather than bytes, even if sizes are specified in bytes. Hence, for example, a three-byte vector would allocate one four-byte cell, and a five-byte vector would allocate two four-byte cells.

```
#define RAW_VECLINK 0
#define RAW_VECSIZE 1
#define RAW_VECDATA 2
```

The **unmark_vecs()** function unmarks all vectors in the vector pool by breaking their "link" field, i.e. setting it to **NIL**. The mark phase of the node collector will later restore the link fields of live vector objects, thereby marking them as "used".

The unmarking function uses the **vecsize()** macro, which was defined earlier in this text, to skip over vectors in the pool. The macro returns the size of a vector in cells given its size in bytes.

The size includes the meta data (link, size) of the vector.

```
void unmark_vecs(void) {
    int     p, k, link;

    p = 0;
    while (p < Freevec) {
        link = p;
        k = Vectors[p + RAW_VECSIZE];
        p += vecsize(k);
        Vectors[link] = NIL;
    }
}
```

The **gcv()** function compacts the vector pool, thereby removing the values of all "dead" objects from the pool. Like the **unmark_vecs()** function, it skips over all vectors in the pool, but at the same times collects all live objects at the beginning of the pool.

Because vector data will move during compaction, each node pointing to a vector has to be adjusted so that it points to the new address of its data. This is what the "link" field is for: the compactor will use this field to locate the node belonging to the data it moves.

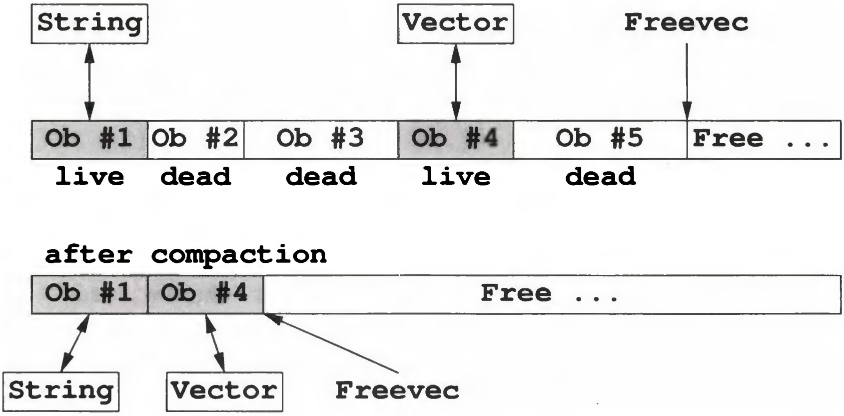


Fig. 16 – Vector pool compaction

After compacting the pool, the **Freevec** variable will be pointed at the first cell after the collected vector data. The area pointed to by **Freevec** is now one single continuous space containing all free vector cells. The process is illustrated in figure 16.

Note that vector pool compaction always includes a node collection, because live vectors will be marked during node collection.

```
int gcv(void) {
    int     v, k, to, from;
    char    buf[100];

    unmark_vecs();
    gc();                   /* re-mark live vectors */
    to = from = 0;
    while (from < Freevec) {
        v = Vectors[from + RAW_VECSIZE];
        k = vecsize(v);
        if (Vectors[from + RAW_VECLINK] != NIL) {
            if (to != from) {
                memmove(&Vectors[to],
                        &Vectors[from],
                        k * sizeof(cell));
                cdr(Vectors[to + RAW_VECLINK]) =
                    to + RAW_VECDATA;
            }
            to += k;
        }
        from += k;
    }
    k = Freevec - to;
    if (GC_verbose) {
        sprintf(buf, "GCV: %d cells reclaimed",
                k);
        prints(buf); nl();
        flush();
    }
    Freevec = to;
```

```
    return k;
}
```

7.5 Vector Allocation

The **newvec()** function allocates a vector of the given type and size, where *type* is the type tag to be placed in the car field of the resulting vector node and *size* is the desired size in bytes.

The function first computes the required size in cells and stores it in the variable *wsize*. When sufficient space is available, it is allocated by moving **Freevec** by *wsize* cells. The new vector is linked to a freshly allocated node of the type *type* and the meta data of the vector is set up by initializing its "link" and "size" fields. Initialization of the data field will be done by functions that use **newvec()** to allocate vector storage. **newvec()** returns the vector node linked to the allocated storage.

Note that the vector data of the returned object *must* be initialized before the next garbage collection takes place or the GC will try to trace random offsets, which will preserve unused space in the best case and crash the collector in the worst case (when values of vector slots are not valid node offsets).

When insufficient space is available, a vector compaction is performed. When space is still insufficient after compaction, the computation is progress is aborted by signaling an error.

```
cell newvec(cell type, int size) {
    cell    n;
    int     v, wsize;

    wsize = vecsize(size);
    if (Freevec + wsize >= NVCELLS) {
        gcv();
        if (Freevec + wsize >= NVCELLS)
            error("newvec: out of vector space",
                  UNDEF);
    }
    v = Freevec;
```

```
    Freevec += wsize;
    n = cons3(type, v + RAW_VECDATA,
              VECTOR_TAG);
    Vectors[v + RAW_VECLINK] = n;
    Vectors[v + RAW_VECSIZE] = size;
    return n;
}
```

7.6 GC Protection

When allocating complex structures consisting of multiple nodes, it is sometimes necessary to protect a node temporarily while allocating a different part of the structure. Consider the expression

```
n = cons(cons(a, b), cons(c, d))
```

which results in the object **((a . b) . (c . d))**. No matter what kind of object a and b are, they will be protected during the application of $cons(a, b)$. However, the result of $cons(a, b)$ will not be protected while $cons(c, d)$ evaluates (or vice versa, if $cons(c, d)$ should be evaluated first).

One of the partial results has to be *protected* explicitly from being reclaimed by the garbage collector while the other one is being allocated. This can be done in two ways: either by binding the object to the GC root **Tmp** or by pushing it to the **Protected** stack, which is a stack whose only purpose is to protect objects from being recycled. Both the code fragment

```
Tmp = cons(a, b);
n = cons(Tmp, cons(c, d));
Tmp = NIL;
```

and the fragment

```
protect(n = cons(a,b));
n = cons(n, cons(c, d));
unprot(1);
```

would serve this purpose. As a rule of thumb, **Tmp** should be used when the allocator is simple, like **cons()**, and only one single value has to be protected. The combination of **prot()** and **unprot()**

should be used when the allocator is complex (like a port allocator or a closure allocator) and/or multiple values must be protected.

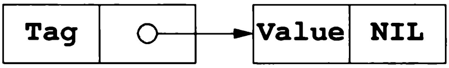
The **protect()** macro pushes its argument to the protection stack, which is a GC root. **unprot()** pops the given number of arguments from the protection stack and returns the last element it removed from the stack.

```
cell    Protected = NIL;
cell    Tmp = NIL;

#define protect(n) \
    (Protected = cons((n), Protected))

cell unprot(int k) {
    cell    n = NIL; /*LINT*/

    while (k) {
        if (NIL == Protected)
            error("unprot: stack underflow",
                  UNDEF);
        n = car(Protected);
        Protected = cdr(Protected);
        k--;
    }
    return n;
}
```

8. High-Level Data Types

This chapter will define the more abstract data types that will be used in the LISP9 system, such as fixnums, characters, strings, vectors, I/O ports, and symbols. It will also implement the *hash table*, a structure that is not visible at LISP level, but will be used internally.

As a reminder, many *atoms* use the layout depicted in figure 2, which shall be shown again here:



The objects implemented in this way are

- The fixnum (small integer)
- The char (character)
- The inport and outport (I/O ports)

Each of them will have the corresponding type tag (such as **T_FIXNUM** or **T_INPORT** in the place of the “tag” and its actual value in the place of the “value”. Both nodes of such atoms are atomic (non-cons) nodes, so the garbage collector will not touch their car parts. Therefore, any value can be stored in their car fields, not just references to other nodes.

The **mkfix()** macro creates a *fixnum* object and stores the value of a C **int** in its value field. The **fixval()** macro extracts that value.

The macros **add_ovfl()** and **sub_ovfl()** return truth (1), if adding/subtracting their **int** arguments a and b would cause an arithmetic overflow. They will be used to make sure that the LISP interpreter will not compute undefined values in math operations. While LISP9 does not provide unbounded integer arithmetic, it will at least signal an error in case of arithmetic overflow.

```
#define mkfix(n) \
    mkatom(T_FIXNUM, mkatom((n), NIL))
```

```
#define fixval(n) (cadr(n))

#define add_ovfl(a,b) \
    ((((b) > 0) && ((a) > INT_MAX - (b))) || \
     (((b) < 0) && ((a) < INT_MIN - (b))))

#define sub_ovfl(a,b) \
    ((((b) < 0) && ((a) > INT_MAX + (b))) || \
     (((b) > 0) && ((a) < INT_MIN + (b))))
```

The **mkchar()** macro creates a *char* object and stores the code point of an (extended) 8-bit ASCII *character* in its value field. **charval()** extracts a code point from a char object.

```
#define mkchar(c) \
    mkatom(T_CHAR, mkatom((c) & 0xff, NIL))

#define charval(n) (cadr(n))
```

The **mkstr()** function creates a *string* object with a length of k characters and optionally initializes it with the content of s. When s is **NULL**, the vector storage of the string object will be cleared by setting all of its bytes to zero. When s is not **NULL**, then **strlen(s)** should be equal to k, or part of the resulting string may be in an undefined state.

Note that LISP9 strings are counted *and* NUL-terminated. Their length is stored in the meta data of the associated vector data and can be extracted using the **stringlen()** macro. They are also NUL-terminated in order to facilitate their use in C functions like **fopen()**, which expect C strings.

When a string of the length 0 is created, **mkstr()** will just return the **Nullstr** object, which is a pre-allocated empty string.

See figure 17 for the layout of string and vector objects.

```
cell    Nullstr = NIL;
```

```
cell mkstr(char *s, int k) {
    cell    n;

    if (0 == k) return Nullstr;
    n = newvec(T_STRING, k+1);
    if (NULL == s) {
        memset(string(n), 0, k+1);
    }
    else {
        memcpy(string(n), s, k);
        string(n)[k] = 0;
    }
    return n;
}
```

All **vector objects** have the structure outlined in figure 17: an atomic node with a type tag in its car field and a reference to a vector pool slot in its cdr field. This vector pool slot will be the first cell of a continuous set of cells that is used to store the value of the vector object.

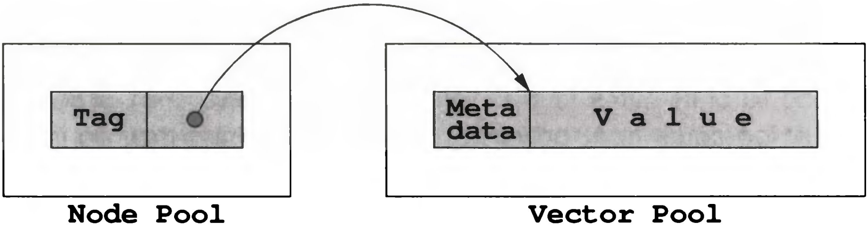


Fig. 17 – Vector object layout

The **mkvec()** function creates a vector object of the type "vector" (irritatingly also called a "vector object" or just a "vector"). Sometimes the term "LISP vector" or "LISP vector object" will be used in order to differentiate between (internal) vectors and LISP vectors. The object returned by **mkvec()** will have k slots. When $k = 0$, **mkvec()** will return **Nullvec**, which is a pre-allocated vector of 0 slots.

```
cell    Nullvec = NIL;
```

```
cell mkvec(int k) {
    cell    n, *v;
    int     i;

    if (0 == k) return Nullvec;
    n = newvec(T_VECTOR, k * sizeof(cell));
    v = vector(n);
    for (i=0; i<k; i++) v[i] = NIL;
    return n;
}
```

The **mkport()** function creates a *port object* of the type *type* and links it to the port *portno*, where *type* is either **T_INPORT** or **T_OUTPORT**, and *portno* is a port handle as delivered by the **newport()** function.

An *I/O port* is an atomic two-node structure, like the fixnum or the char. It stores the port number *portno* in its “value” field. The **portno()** macro, which has been defined earlier in this text, can be used to extract the port number.

Note that the port *portno* has to be locked while allocating the port atom. If it was not, it could be closed prematurely when **mkatom()** or **cons3()** triggers a garbage collection.

```
cell mkport(int portno, cell type) {
    cell    n;
    int     pf;

    pf = Port_flags[portno];
    Port_flags[portno] |= LOCK_TAG;
    n = mkatom(portno, NIL);
    n = cons3(type, n, ATOM_TAG|PORT_TAG);
    Port_flags[portno] = pf;
    return n;
}
```

The following functions implement the *hash table*. A hash table is a structure that is used to associate arbitrary values ("keys") with other arbitrary values ("data") in such a way that data associated with a given key can be retrieved quickly. It can be thought of as a more efficient form of the association list.

The implementation presented here offers dynamically growing hash tables and functions for creating new tables as well as for looking up, adding, and removing associations.

The **htsize()** function computes the size of a vector that will be used for storing associations. Hash tables with a weak hash function work most efficiently when their size is a prime number. **htsize()** will return a prime number that is larger than the requested hash table size (unless the table is larger than the largest number returned by the function, but this should never happen).

```
int htsize(int n) {
    if (n < 47) return 47;
    if (n < 97) return 97;
    if (n < 199) return 199;
    if (n < 499) return 499;
    if (n < 997) return 997;
    if (n < 9973) return 9973;
    if (n < 19997) return 19997;
    return 39989;
}
```

The **mkht()** function creates an empty hash table of (at least) the requested size, k. The general structure of a hash table is shown in figure 18.

Basically, the hash table is a pair with a fixnum in its car field and a vector in its cdr field. The fixnum ("count") equals the number of associations currently stored in the hash table. Each vector slot contains an *association list* of the form

```
((key₁ . value₁)
  ...
```

```
 (key_n . value_n))
```

where each **(key . value)** pair is an individual association. An empty association list, like any empty list, is equal to **NIL**, in this case indicating that a hash table slot is currently unused.

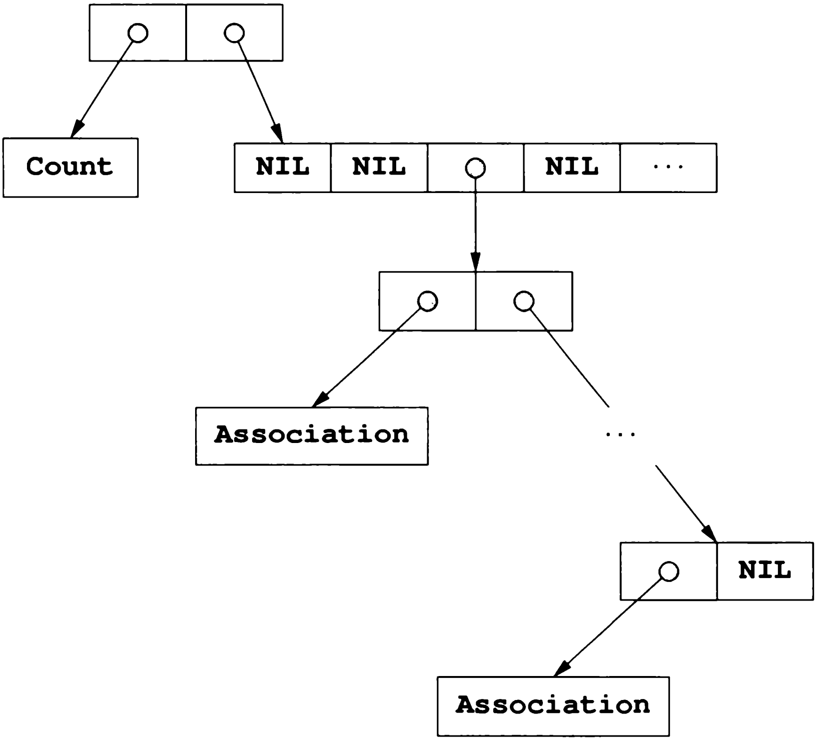


Fig. 18 – Hash table structure

Elements can be located in the association lists of the slots by using LISP's **assoc** function. Although **assoc** has linear run time, the lists are usually *very* short, because associations are quite homogeneously distributed across slots, so elements can still be considered to be accessible in constant time.

```
cell mkht(int k) {
    cell    n;

    n = mkfix(0); /* mutable, can't use Zero */
    protect(n);
    n = cons(n, mkvec(k));
```

```
    unprot(1);
    return n;
}
```

The following macros access elements of a hash table: **htlen()** returns the length of the vector containing the table slots, **htelts()** returns the current association count of the table, and **htdata()** returns the vector in which the associations are stored. **htslots()**, finally, returns a *pointer* to the slots of the table, so that a C function can access them without going through the **vector()** abstraction for each single element.

```
#define htlen(d)     veclen(cdr(d))
#define htelts(d)    fixval(car(d))
#define htdata(d)    cdr(d)
#define htslots(d)   vector(cdr(d))
```

The **hash()** function computes the hash value of the characters in the string s modulo k, where k is the number of slots in the hash table for which the hash value is being computed. This value will be used to locate the hash table slots of strings and symbol names.

```
uint hash(byte *s, uint k) {
    uint    h = 0xabcd;

    while (*s) h = ((h << 5) + h) ^ *s++;
    return h % k;
}
```

The **obhash()** function computes hash values for a variety of different objects. See figure 19 for details. The expression $x \ rem \ y$ in the figure denotes the division remainder of the truncated integer division x/y.

Type of x	**Hash Value of** x
char	$x\ rem\ k$
fixnum	$\lvert fixval(x) \rvert\ rem\ k$
special object	$\lvert x \rvert\ rem\ k$
string	$hash(x)\ rem\ k$
symbol	$hash(x)\ rem\ k$
all others	0

Fig. 19 – Hash value computation

```
uint obhash(cell x, uint k) {
    if (specialp(x))
        return abs(x) % k;
    if (symbolp(x))
        return hash(symname(x), k);
    if (fixp(x))
        return abs(fixval(x)) % k;
    if (charp(x))
        return charval(x) % k;
    if (stringp(x))
        return hash(string(x), k);
    return 0;
}
```

The expression $match(a, b)$ will return truth, if a and b are equal in the sense of **equal**, i.e. if

- a and b are equal in the sense of **eq**
- both a and b are characters and indicate the same code point
- both a and b are fixnums are have the same value
- both a and b are strings and their characters are pairwise equal in the sense of **c=**

In addition, the function will return truth, if both a and b are symbols and their names (strings) match, *even if a and b are not the same symbol.* This extension is necessary, because hash tables will be used exactly for the task of *making* symbols identical, i.e. the symbol table of LISP9 will be a hash table. (See

the **intern** and **symref** functions for details).

```
int match(cell a, cell b) {
    int     k;

    if (a == b) {
        return 1;
    }
    if (fixp(a) && fixp(b)) {
        return fixval(a) == fixval(b);
    }
    if (charp(a) && charp(b)) {
        return charval(a) == charval(b);
    }
    if (symbolp(a) && symbolp(b)) {
        k = symlen(a);
        if (symlen(b) != k) return 0;
        return memcmp(symname(a),
                      symname(b), k) == 0;
    }
    if (stringp(a) && stringp(b)) {
        k = stringlen(a);
        if (stringlen(b) != k) return 0;
        return memcmp(string(a),
                      string(b), k) == 0;
    }
    return 0;
}
```

The **htgrow()** function adds new slots to an existing hash table d (where “d” means “dictionary”, because that is what hash tables are used for in LISP9). When the number of elements in a given hash table becomes larger than its number of slots, hash collisions may increase to a level where the hash table becomes inefficient. At this point, new slots will be added to the table.

htgrow() first creates a new table of the next larger size than the existing one and then copies all associations from the old table to

the new one. This step involves the recalculation of all hash slots, because hash values depend on the size of the table. Finally, the function *replaces* the data part of the old table by the data part of the new table, thereby extending the table d destructively.

```
void htgrow(cell d) {
    int     nk, i, h, k;
    cell    nd, e, n;

    k = htlen(d);
    nk = 2 * htlen(d);
    nd = mkht(nk);
    protect(nd);
    nk = htlen(nd);
    for (i = 0; i < k; i++) {
        for (e = htslots(d)[i];
             e != NIL;
             e = cdr(e))
        {
            h = obhash(caar(e), nk);
            n = cons(car(e), htslots(nd)[h]);
            htslots(nd)[h] = n;
        }
    }
    htdata(d) = htdata(nd);
    unprot(1);
}
```

The **htlookup()** function looks up an association with the key k in the hash table d. If such an association exists, it returns it. When no matching association is found, the function returns **UNDEF**.

```
int htlookup(cell d, cell k) {
    cell    x;
    int     h;

    h = obhash(k, htlen(d));
```

```
    x = htslots(d)[h];
    while (x != NIL) {
        if (match(caar(x), k)) return car(x);
        x = cdr(x);
    }
    return UNDEF;
}
```

The **htadd()** function adds an association with the key k and value v to the hash table d. It will never replace any existing association, but always add a new one. The new association is added by modifying the table.

```
void htadd(cell d, cell k, cell v) {
    cell    e;
    int     h;

    Tmp = k;
    protect(v);
    protect(k);
    Tmp = NIL;
    if (htelts(d) >= htlen(d))
        htgrow(d);
    h = obhash(k, htlen(d));
    e = cons(k, v);
    e = cons(e, htslots(d)[h]);
    htslots(d)[h] = e;
    htelts(d)++;
    unprot(2);
}
```

The **htrem()** function removes an association with the key k from the hash table d. The association is deleted by modifying the table.

```
cell htrem(cell d, cell k) {
    cell    *x, *v;
```

```
    int    h;

    h = obhash(k, htlen(d));
    v = htslots(d);
    x = &v[h];
    while (*x != NIL) {
        if (match(caar(*x), k)) {
            *x = cdr(*x);
            htelts(d)--;
            break;
        }
        x = &cdr(*x);
    }
    return d;
}
```

The *symbol table* (*SYMTAB*) of a LISP system is a collection of *symbol* atoms that is known to the system. The primary function of the symbol table is to provide *symbol identity*, i.e. to make sure that all instances of the same symbolic name, when read by the *reader*, actually refer to the same symbol. That is, it makes sure that

```
(eq 'a 'a)
```

will always return truth, for any symbolic name a.

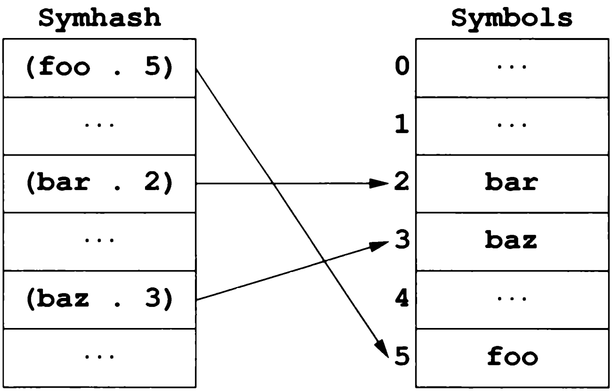


Fig. 20 – Symbol table structure

The LISP9 system uses two structures for mapping symbols to

symbol table slots and vice versa: The hash table **Symhash** maps symbolic names to SYMTAB slots, and the vector **Symbols** maps SYMTAB slot numbers to symbolic names. (See figure 20.) The **Symhash** table also maps symbols to themselves (using the key as value), thereby implementing symbol identity.

Symhash is used by the reader to create references to unambiguous symbols and by the compiler to create debug information. **Symbols** is used for error reporting, because references to symbols themselves will be lost during compilation, but symbol table slot numbers are carried by some abstract machine instructions.

In principle, a *reverse lookup* of a slot number could be performed in **Symhash**, which would save space, but diminish performance. While performance is not important in error reporting, it may be more critical in other applications, like a disassembler for abstract machine code.

The variable **Symptr** points to the next free slot in the **Symbols** vector. The **Symbols** vector, like the corresponding hash table, will grow on demand.

```
cell    Symhash = NIL;
cell    Symbols = NIL;
int     Symptr = 0;
```

The **mksym()** function returns a new symbol, but *does not intern it*! This means that the resulting symbol with be neither unique nor unambiguous. The only LISP9 function that creates uninterned symbols is **gensym**, which will be explained later.

```
cell mksym(char *s, int k) {
    cell    n;

    n = newvec(T_SYMBOL, k+1);
    strcpy((char *) symname(n), s);
    return n;
}
```

The **findsym()** function looks up the symbol name *s* in the symbol table. If such a symbol exists, it returns the unique atom representing that symbol. If no symbol with the given name exists, the function returns **NIL**.

```
cell findsym(char *s) {
    cell    y;

    y = mksym(s, strlen(s));
    y = htlookup(Symhash, y);
    if (y != UNDEF) return car(y);
    return NIL;
}
```

The **intern()** function *interns* the symbol *y* by adding it to both the hash table **Symhash** and the vector **Symbols**. It returns the interned symbol.

When the next free slot (**Symptr**) is outside of the **Symbols** vector, the function will extend the vector by **CHUNKSIZE** elements. Because vectors cannot grow in the memory model used here, this is done by creating a new vector and copying the existing symbols to the new vector.

```
cell intern(cell y) {
    cell    n, *vn, *vs;
    int     i, k;

    htadd(Symhash, y, mkfix(Symptr));
    k = veclen(Symbols);
    if (Symptr >= k) {
        n = mkvec(k + CHUNKSIZE);
        vs = vector(Symbols);
        vn = vector(n);
        for (i=0; i<k; i++) vn[i] = vs[i];
        Symbols = n;
```

```
    }
    vector(Symbols)[Symptr] = y;
    Symptr++;
    return y;
}
```

The **symref()** function returns a reference to an interned symbol with the name *s*. The symbol may or may not exist and may or may not be interned when **symref()** is called.

When the symbol exists and is interned, a reference to it will be returned. When the symbol exists and is not interned, a *new instance* of the symbol will be created and interned. When the symbol does not exist at all, it will be created and interned.

```
cell symref(char *s) {
    cell    y, new;

    y = findsym(s);
    if (y != NIL) return y;
    new = mksym(s, strlen(s));
    return intern(new);
}
```

9. Some Useful List Functions

This short chapter defines some list processing functions that will be useful for implementing the more complex algorithms in the later chapters, for instance in the reader and the compiler.

The **reconc()** function reverses its first argument, n, and concatenates it to its second argument, m. It is a very natural function, because reversing n is maybe the most straightforward way to concatenate it to m. The corresponding LISP algorithm would be

```
(do ((n n (cdr n))
     (m m (cons (car n) m)))
    ((null n) m))
```

The C version is a straightforward translation of this expression. Passing a dotted list in n will signal an error.

```
cell reconc(cell n, cell m) {
    while (n != NIL) {
        if (atomp(n))
            error("reconc: dotted list", n);
        m = cons(car(n), m);
        n = cdr(n);
    }
    return m;
}
```

---

reverse() is easily implemented by passing **NIL** as the second argument to **reconc()**.

```
#define reverse(n) reconc((n), NIL)
```

---

nreconc() is like **reconc()**, but reverses and concatenates its arguments destructively. It does not allocate any nodes.

```
cell nreconc(cell n, cell m) {
    cell    h;

    while (n != NIL) {
        if (atomp(n))
            error("nreconc: dotted list", n);
        h = cdr(n);
        cdr(n) = m;
        m = n;
        n = h;
    }
    return m;
}
```

nreverse() is to **nreconc()** as **reverse()** is to **reconc()**.

```
#define nreverse(n) nreconc((n), NIL)
```

The **conc()** function concatenates its arguments *a* and *b*. It is basically implemented as

```
(reconc (rever a) b)
```

here, but without the error checking, because it will always be called with the proper structure in *a*.

The LISP function of the same name is variadic. It will be defined in a later chapter of this book.

```
cell conc(cell a, cell b) {
    cell    n;

    a = reverse(a);
    protect(a);
    n = b;
    while (a != NIL) {
        n = cons(car(a), n);
        a = cdr(a);
    }
```

```
    unprot(1);
    return n;
}
```

nconc() is like **conc()**, but concatenates its arguments destructively and without allocating any nodes.

```
cell nconc(cell a, cell b) {
    cell    n;

    n = a;
    if (NIL == a) return b;
    while (cdr(a) != NIL) a = cdr(a);
    cdr(a) = b;
    return n;
}
```

10. High-Level Port I/O

The **newport()** function allocates an unused *port number* and returns it. When no ports are free, it initiates a garbage collection to free one. If that fails, it returns −1, indicating failure.

```
int newport(void) {
    int    i, n;

    for (n=0; n<2; n++) {
        for (i=0; i<NPORTS; i++) {
            if (NULL == Ports[i])
                return i;
        }
        if (0 == n) gc();
    }
    return -1;
}
```

The **open_inport()** function opens the file *path* for input, associates it with a port number and returns that port number. When the file cannot be opened or no free port is available, the function returns −1 to signal failure.

open_outport() is like **open_inport()**, but opens the given file for output. If the specified file already exists, it will be truncated to zero length. When the second argument of **open_outfile()**, *append*, is non-zero, an existing file will *not* be truncated and all output sent to the port is appended to the output file instead.

```
int open_inport(char *path) {
    int    i;

    i = newport();
    if (i < 0) return -1;
    Ports[i] = fopen(path, "r");
    if (NULL == Ports[i]) return -1;
```

```
    return i;
}

int open_outport(char *path, int append) {
    int    i;

    i = newport();
    if (i < 0) return -1;
    Ports[i] = fopen(path, append? "a": "w");
    if (NULL == Ports[i]) return -1;
    return i;
}
```

The **set_inport()** function makes *port* the new current input port and returns the input port that was in effect previously. **set_outport()** does the same for the current output port.

```
cell set_inport(cell port) {
    cell    p = Inport;

    Inport = port;
    return p;
}

int set_outport(int port) {
    int    p = Outport;

    Outport = port;
    return p;
}
```

The **close_port()** function closes an input or output file and releases the associated port number, so that subsequent **open_infile()** or **open_outfile()** operations can reuse it. The function will ignore invalid port numbers and port numbers associated with closed/unused ports.

```
void close_port(int port) {
    if (port < 0 || port >= NPORTS)
        return;
    if (NULL == Ports[port]) {
        Port_flags[port] = 0;
        return;
    }
    fclose(Ports[port]);
    Ports[port] = NULL;
    Port_flags[port] = 0;
}
```

The **reset_stdports()** function clears the error flag on the standard I/O **FILE**s and resets the standard I/O ports **Inport**, **Outport**, and **Errport** to their default values. When the LISP interpreter is connected to a terminal, calling this function has the effect of returning control to the terminal in case one of the current I/O ports was redirected.

```
void reset_stdports(void) {
    clearerr(stdin);
    clearerr(stdout);
    clearerr(stderr);
    Inport = 0;
    Outport = 1;
    Errport = 2;
}
```

The **lock_port()** function *locks* the I/O port whose number is specified in *port*. It returns 0 on success and −1 when an invalid port number is passed to it.

A locked port will never be finalized by the garbage collector. Hence locking a port is the same as *GC-protecting* a node. In fact, a port can be protected by protecting its associated port atom, but the **lock_port()** function can GC-protect a port that is not or not yet associated with any node.

The **unlock_port()** function, unsurprisingly, unlocks the given port.

```
int lock_port(int port) {
    if (port < 0 || port >= NPORTS)
        return -1;
    Port_flags[port] |= LOCK_TAG;
    return 0;
}

int unlock_port(int port) {
    if (port < 0 || port >= NPORTS)
        return -1;
    Port_flags[port] &= ~LOCK_TAG;
    return 0;
}
```

11. The Global Environment

The LISP9 system has two *global environments* that bind symbolic names to locations. One is a *deep-binding* environment in which new bindings will be created and which C functions will use to look up values. The other one is a *shallow-binding* environment, which is used by compiled programs to look up and modify values of variables.

Deep-binding environments are *association lists* that associate symbols with *boxes*. A box is a list of a single element in which values will be stored. Lookup is linear, but this is good enough for the rare and not very time critical occasions where the interpreter has to access the environment.

The run time environment, which is shallow-binding with constant time access, will be introduced in a later chapter.

The variable **Glob** is bound to the deep-binding global environment.

```
cell    Glob = NIL;
```

The **bindnew()** function creates a new *variable* by binding the symbol v to a box containing the value (argument) a. It adds the new binding to the head of **Glob**.

```
void bindnew(cell v, cell a) {
    cell    n;

    n = cons(a, NIL);
    n = cons(v, n);
    Glob = cons(n, Glob);
}
```

The **assq()** function is (almost) equal to the Scheme function of the same name. It looks up an association with the key x in the

association list *a* and returns it. When no such association exists, it returns NIL. **assq** assumes that *a* is a proper association list.

```
int assq(cell x, cell a) {
    for (; a != NIL; a = cdr(a))
        if (caar(a) == x) return car(a);
    return NIL;
}
```

The **bindset()** function stores the value *a* in the box associated with the symbol *v* in the deep-binding global environment. When no matching association exists, the function does nothing.

```
void bindset(cell v, cell a) {
    cell    b;

    b = assq(v, Glob);
    if (b != NIL) cadr(b) = a;
}
```

12. The Reader

The LISP *reader* is the part of a LISP system that reads the *external representations* of *data objects* from an input port and returns the corresponding objects. Its is available to LISP programs via the **read** subroutine.

The external representation of an object is what the user would type in in order to create an object of the desired type. For instance, the empty list and "false" truth value have the external representation **nil**, the fixnum 123 has the external representation **123**, and the ASCII character **'A'** has the external representation **#\A**. See figure 21 for more examples.

Type	Examples
special	`nil t`
symbol	`foo-bar + a/b`
fixnum	`123 0 -1 #16rFFF`
char	`#\a #\sp #\\33`
string	`"" "foo\n" "\"hi\""`
vector	`#() #(a) #(a #(b) c)`
pair	`(a . b)`
list	`() (a b) ((a)) (a b c)`
dotted list	`(a . b) (a b c . d)`

Fig. 21 – External representation examples

The external representation of an object does not have to be unique. For instance, all of the following external forms represent **nil**:

```
nil   Nil   NIL   niL
```

all of the following represent the number 123:

```
123   0123   00123   +123   +000123   #16r7B
```

and all of the following represent the list **(a b c)**:

```
(a b c)
(a b . (c))
(a . (b . (c . nil)))
```

```
(a  b  c)
(a
   b
     c)
```

However, each data object has a *canonical external representation* that will be printed by the **prin** function – which can be considered to be the inverse function of **read**.

The reader will also accept (and expand) some abbreviations, like **'foo** instead of **(quote foo)**, **@bar** instead of **(qquote bar)**, and **,l baz** instead of **(load "baz.ls9")**, etc.

The reader presented here will accept all of the examples listed above and more. Do not expect too much wisdom, though: the implementation is merely an ad-hoc *recursive descent parser* that will accept all possible external forms of the types listed in figure 21 plus abbreviations and *comments*, which are a bit special in LISP9.

The following variables are bound to predefined symbols, like LISP9 keywords and the names of top-level (special) variables. Most of the variables bind to symbols of the corresponding name. For instance, **S_apply** is bound to the symbol **apply**. Note that the term "is bound to a symbol" actually means "bound to a storage location holding a reference to a symbol", but the former will be used when the difference would be of no consequence.

Symbol	Variable
`S_errtag`	`*errtag*`
`S_errval`	`*errval*`
`S_ifstar`	`if*`
`S_imagefile`	`*imagefile*`
`S_quiet`	`*quiet*`
`S_starstar`	`**`

Fig. 22 – Predefined symbols and their variables (exceptions)

There are a few exceptions to the above rule, though, which are listed in figure 22.

```
cell    S_apply, S_comment, S_def, S_defmac,
```

```
         S_defun, S_errtag, S_errval, S_if,
         S_ifstar, S_imagefile, S_labels,
         S_lambda, S_macro, S_prog, S_quiet,
         S_quote, S_qquote, S_starstar, S_splice,
         S_setq, S_start, S_unquote;
```

The following variables represent *primitive functions* in the sense that these functions are predefined at C level. All of these functions will be inlined by the compiler. The naming follows the same pattern as described above: the **P_** prefix is removed. In addition, each '_' character in the remaining name is replaced by a **'-'** and the following translations will be performed: "**less**"→"<", "**lteq**"→"<=", "**equal**"→"=", "**grtr**"→">", and "**gteq**"→">=".

So, for instance, **P_open_inport** will be bound to **open-inport** and **P_siequal** to **si=**.

```
cell     P_abs, P_alphac, P_atom, P_bitop,
         P_caar, P_cadr, P_car, P_catchstar,
         P_cdar, P_cddr, P_cdr, P_cequal,
         P_cgrtr, P_cgteq, P_char, P_charp,
         P_charval, P_cless, P_close_port,
         P_clteq, P_cmdline, P_conc, P_cons,
         P_constp, P_ctagp, P_delete, P_div,
         P_downcase, P_dump_image, P_eofp, P_eq,
         P_equal, P_gc, P_error, P_errport,
         P_eval, P_existsp, P_fixp, P_format,
         P_funp, P_gensym, P_grtr, P_gteq,
         P_inport, P_inportp, P_less, P_liststr,
         P_listvec, P_load, P_lowerc, P_lteq,
         P_max, P_min, P_minus, P_mkstr, P_mkvec,
         P_mx, P_mx1, P_nconc, P_nreconc, P_not,
         P_null, P_numeric, P_numstr, P_obtab,
         P_open_infile, P_open_outfile,
         P_outport, P_outportp, P_pair, P_peekc,
         P_plus, P_prin, P_princ, P_quit, P_read,
         P_readc, P_reconc, P_rem, P_sconc,
         P_sequal, P_set_inport, P_set_outport,
```

```
        P_setcar, P_setcdr, P_sfill, P_sgrtr,
        P_sgteq, P_siequal, P_sigrtr, P_sigteq,
        P_siless, P_silteq, P_sless, P_slteq,
        P_sref, P_sset, P_ssize, P_stringp,
        P_strlist, P_strnum, P_substr, P_subvec,
        P_symbol, P_symbolp, P_symname,
        P_symtab, P_syscmd, P_throwstar,
        P_times, P_untag, P_upcase, P_upperc,
        P_veclist, P_vconc, P_vectorp, P_vfill,
        P_vref, P_vset, P_vsize, P_whitec,
        P_writec;
```

The **Intr** ("interrupt") flag, which will be defined later, is used to signal the reader that the user wishes to abort an input operation at the REPL. It will be set asynchronously when a *SIGINT* signal is received.

```
volatile int     Intr;
```

The variable **Inlist** keeps track of the current list nesting level. This is necessary, because certain characters, like the the dot ('**.**') and closing parenthesis ('**)**'), may only appear in list/pair contexts.

The variable **Quoting** indicates that a **quote** or **qquote** expression is currently being read. Quotation is special, because the comma ('**,**') character has different meanings, depending on its context. Inside of quasiquotation, **,x** is short for **(unquote x)** and at the top level it introduces an abbreviation, such as **,l file**.

```
int    Inlist = 0;
int    Quoting = 0;
```

The **octalp()** macro tests whether a character represents an octal digit and the **octchar()** function converts the representation of an octal number in the string s to an integer and returns it. The representation in s is only considered to be valid under certain constraints: there may not follow any characters after the octal

characters, and the resulting integer must not be greater than 255. When the result is invalid, **octchar()** returns −1.

The **octchar()** function is used to parse *octal char names*, such as **#\\33** or **#\\377**. Such character names are used to represent code points that have no specific glyph associated with them and therefore cannot be entered using a keyboard.

```
#define octalp(c) \
    ('0' == (c) || '1' == (c) || '2' == (c) || \
     '3' == (c) || '4' == (c) || '5' == (c) || \
     '6' == (c) || '7' == (c))

int octchar(char *s) {
    int    v = 0;

    if (!octalp(*s)) return -1;
    while (octalp(*s)) {
        v = 8*v + *s - '0';
        s++;
    }
    return (*s || v > 255)? -1: v;
}
```

The **symbolic()** macro tests whether the given character c is a character that may appear in a symbolic name. The test for c being non-zero stops **strchr()** from finding a NUL character in the string of special characters, i.e. it makes NUL a non-symbolic character.

```
#define symbolic(c) \
    (isalpha(c) || \
     isdigit(c) || \
     (c && strchr("!$%^&*-/_+=~.?<>:", c)))
```

The constants **LP** and **RP** are used instead of the parenthesis characters in order to keep the parenthesis balancing happy in text

editors.

```
#define LP  '('
#define RP  ')'
```

The **strcmp_ci()** function is like **strcmp()**, but compares the strings $s1$ and $s2$ without distinguishing upper and lower case. It will be used to compare symbolic *char names*, as used in literals like **#\sp**, thereby allowing to write **#\Sp** or **#\SP**.

```
int strcmp_ci(char *s1, char *s2) {
    int     c1, c2;

    while (1) {
        c1 = tolower((int) *s1++);
        c2 = tolower((int) *s2++);
        if (!c1 || !c2 || c1 != c2)
            break;
    }
    return c1-c2;
}
```

The variable *Readerr* ("reader error"), like **Intr**, makes the reader discard all subsequent input and return an unspecific result as fast as possible. Their difference is purely conceptional. While **Intr** is used signal to an asynchronous event, **Readerr** indicates that an error (like a syntax error) has occurred while reading an object. When this happens, **Readerr** will point to a message describing the cause of the error.

The LISP9 reader can not only read ports but also strings. The **Readerr** variable is used to indicate failure to read an object from a string.

```
char    *Readerr = NULL;
```

The **rderror()** function handles errors while reading the external

representation of an object. When the source of input is a port, an error will be signalled and otherwise the **Readerr** variable will be pointed at the argument s, which contains the error message.

All reader functions will perform error handling via **rderror()**.

```
void rderror(char *s, cell x) {
    if (NULL == Instr) error(s, x);
    Readerr = s;
}
```

The **rdchar()** function reads a *char* literal and returns a char object. Char literals have one out of the following three forms

- **#\c**, where "c" is a glyph representing a char
- **#\name**, where "name" is a symbolic char name
- **#\\octal**, where "octal" is a code point in octal notation (note the double backslash!)

Valid glyphs are those defined in the ASCII alphabet, valid symbolic char names are **ht** ("horizontal tabulator"), **nl** ("newline") and **sp** ("space", blank). The newline character may translate to multiple characters during output, depending on the operating environment. Octal notation is an octal integer in the range from 0 to 377_8.

When **rdchar()** fails to extract a valid char literal from its input source, it will signal an error.

```
cell rdchar(void) {
    char    name[TOKLEN+1];
    int     i, c, v;

    c = readc();
    name[0] = c;
    c = readc();
    for (i=1; i<TOKLEN; i++) {
        if (Intr || Readerr) return NIL;
        if (!isalpha(c) && !isdigit(c)) break;
```

```
        name[i] = c;
        c = readc();
    }
    name[i] = 0;
    rejectc(c);
    if (TOKLEN == i)
        rderror("char name too long",
            mkstr(name, strlen(name)));
    if (!strcmp_ci(name, "ht"))
        return mkchar(9);
    if (!strcmp_ci(name, "nl"))
        return mkchar(10);
    if (!strcmp_ci(name, "sp"))
        return mkchar(' ');
    v = octchar(&name[1]);
    if ('\\' == *name && v >= 0)
        return mkchar(v);
    if (i != 1)
        rderror("bad character name",
                mkstr(name, strlen(name)));
    return mkchar(name[0]);
}
```

The **rdlist()** function reads a *pair*, *list*, or *dotted list* from its input source and returns the root node of its internal representation. It also translates the *empty list* **()** to **nil**. The resulting object will be *immutable*, i.e. all cons nodes in the "spine" of the list will have the **CONST_TAG** bit set. See figure 23 for the *spine* of a list.

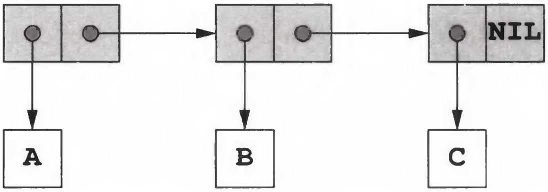


Fig. 23 – The spine of the list (a b c), shown in grey

The **rdlist()** function will signal an error when the end of its input is encountered while there are still any parentheses left open or

when it finds an incomplete or malformed dotted pair, like **(. a)**, **(a .)**, or **(a . b c)**.

```
cell    xread2(void);

cell rdlist(void) {
    cell        n, a, p;
    cell        new;
    static char badpair[] = "malformed pair";

    Inlist++;
    n = xread2();
    if (RPAREN == n) {
        Inlist--;
        return NIL;
    }
    p = NIL;
    a = cons3(NIL, NIL, CONST_TAG);
    protect(a);
    while (n != RPAREN) {
        if (Intr || Readerr) {
            unprot(1);
            return NIL;
        }
        if (EOFMARK == n)  {
            rderror("missing ')'", UNDEF);
            return NIL;
        }
        else if (DOT == n) {
            if (NIL == p) {
                unprot(1);
                rderror(badpair, UNDEF);
                return NIL;
            }
            n = xread2();
            cdr(p) = n;
            if (RPAREN == n ||
                xread2() != RPAREN)
```

```
                {
                    unprot(1);
                    rderror(badpair, UNDEF);
                    return NIL;
                }
                Inlist--;
                return unprot(1);
            }
            car(a) = n;
            p = a;
            n = xread2();
            if (n != RPAREN) {
                Tmp = n;
                new = cons3(NIL, NIL, CONST_TAG);
                Tmp = NIL;
                cdr(a) = new;
                a = cdr(a);
            }
        }
    Inlist--;
    return unprot(1);
}
```

The **rdvec()** function reads a vector literal and returns an immutable vector. It first reads a list and then converts it to a vector using **listvec()**, which will be defined later. The *veclit* flag of **veclist()** indicates that the result shall be immutable and that the function is being called in order to convert a vector literal rather than an argument of the **listvec** function of LISP9.

```
cell    listvec(cell x, int veclit);

cell rdvec(void) {
    return listvec(rdlist(), 1);
}
```

The **pos()** function returns the first offset at which the character *p*

is found in the string s, or -1 if s does not contain p. It will be used to compute the values of non-decimal digits. E.g.

```
pos('c', "0123456789abcdef")
```

would return 12, the value of the hexa-decimal digit c.

```
int pos(int p, char *s) {
    int     i;

    i = 0;
    for (; *s; s++) {
        if (p == *s) return i;
        i++;
    }
    return -1;
}
```

The **scanfix()** function attempts to interpret the string s as a *fixnum* to the base of r. No leading or trailing noise characters are allowed. The function accepts a leading '**+**' or '**–**' sign and any number of digits of the given base. Digits with a base above 10 (decimal ten) are represented by Latin letters. Consequently, the largest supported base is 36. Also, r must not be smaller than 2.

When the *of* ("overflow") flag is set to a non-zero value, the **scanfix()** function will signal an error when the resulting numeric value would cause an integer overflow. When $of = 0$, **NIL** will be returned and no error will be signalled. This option is used to implement the **strnum** ("string to number") function of LISP9.

Upon success **scanfix()** will return a fixnum object.

```
cell scanfix(char *s, int r, int of) {
    int     v, w, g, i;
    char    *p;
    char    d[] = "0123456789abcdefgh"
                  "ijklmnopqrstuvwxyz";

    g = 1;
```

```
    p = s;
    if ('+' == *p) {
        p++;
    }
    else if ('-' == *p) {
        p++;
        g = -1;
    }
    v = 0;
    while (*p) {
        i = pos(tolower(*p), d);
        if (i < 0 || i >= r) return NIL;
        w = v;
        if (v > INT_MAX/r ||
            (v > 0 && add_ovfl(v*r, i)) ||
            (v < 0 && sub_ovfl(v*r, i)))
        {
            if (!of) return NIL;
            rderror("fixnum too big",
                    mkstr(s, strlen(s)));
        }
        else if (v < 0)
            v = v*r - i;
        else
            v = v*r + i;
        p++;
        if (g) v *= g;
        g = 0;
    }
    if (g) return NIL;
    return mkfix(v);
}
```

The **rdsymfix()** function reads a *fixnum object or* a *special object or* a *symbol* and returns its internal representation. The *c* argument contains the first character of the object to be read, *r* is the base for reading fixnum literals, and *sym* indicates that the function is allowed to return a symbol (or special object) – which it

is not when called by **rdfix()**, below.

Because the special objects **nil** and **t** have the outer appearance of symbols, they will also be read by **rdsymfix()**. When reading them, the function will return the corresponding internal forms of the special objects, **NIL** and **TRUE**.

```
cell rdsymfix(int c, int r, int sym) {
    char    name[TOKLEN+1];
    int     i;
    cell    n;

    for (i=0; i<TOKLEN; i++) {
        if (!symbolic(c)) break;
        name[i] = tolower(c);
        c = readc();
    }
    name[i] = 0;
    rejectc(c);
    if (TOKLEN == i)
        rderror("symbol or fixnum too long",
                mkstr(name, strlen(name)));
    n = scanfix(name, r, 1);
    if (n != NIL) return n;
    if (!sym)
        rderror("invalid digits after #radixR",
                mkstr(name, strlen(name)));
    if ('t' == name[0] && 0 == name[1])
        return TRUE;
    if (!strcmp(name, "nil"))
        return NIL;
    return symref(name);
}
```

The **rdfix()** function will read a fixnum literal of the form

#*base*R *sign digits*

where *base* is the base of the number to read, specified as a

decimal number on its own, and *digits* are the digits of the number to read. Everything following the **'R'** is passed to **rdsymfix()** for conversion. Subsequently, the *sign* character is optional. The argument *c* holds the first character of *base*; the leading **'#'** character has already been consumed by the calling function, **xread()**.

```
cell rdfix(int c) {
    int     r;

    r = 0;
    while (isdigit(c)) {
        r = r*10 + c - '0';
        c = readc();
    }
    if (c != 'r')
        rderror("'R' expected after #radix",
                UNDEF);
    if (r < 2 || r > 36)
        rderror("bad radix in #radixR",
                mkfix(r));
    c = readc();
    return rdsymfix(c, r, 0);
}
```

The **rdstr()** function reads a *string literal* and returns its internal representation. The initial quote ("**"**") character has already being consumed when the function is called. It collects characters until it finds another quote character. The collected characters form the value of the string.

A character following a backslash ('\') is treated in a special way: "\"" inserts a quote character in the string without delimiting it and "\\" inserts a backslash. Furthermore, the following translations will be performed: "**\n**"→**#\nl** and "**\t**"→**#\ht**.

Like in char literals, octal numbers following a backslash can be used to insert any char value. However, the octal number may never be longer than three characters. This is necessary to allow

for the formulation of sequences such as

`"0\0002"`

which consists of the character “**0**”, followed by a **NUL** character, followed by the character “**2**”. Without the limitation to three characters, the above string would translate to a two-character string containing the character “**1**” and the code point 2 (ASCII **STX**), parsed from the octal sequence **0002**. In this case there would be no way to express the string consisting of “**1**”, NUL, and “**2**”.

The **rdstr()** function will signal an error, if the end of input is encountered inside of the string literal to parse or if it detects an invalid backslash sequence.

```
cell rdstr(void) {
    char    name[TOKLEN+1];
    int     i, j, c, u, v;
    cell    n;

    c = readc();
    u = 0;
    for (i=0; i<TOKLEN; i++) {
        if (Intr || Readerr) return NIL;
        if ('"' == c) break;
        if ('\n' == c) Line++;
        if (EOF == c)
            rderror("EOF in string", UNDEF);
        if ('\\' == c) {
            c = readc();
            if ('\\' == c || '"' == c) {
                /**/
            }
            else if ('t' == c) {
                c = '\t';
            }
            else if ('n' == c) {
                c = '\n';
            }
```

```
            else if (octalp(c)) {
                v = 0;
                j = 0;
                while (j < 3 && octalp(c)) {
                    v = v * 8 + c-'0';
                    c = readc();
                    j++;
                }
                rejectc(c);
                if (v > 255)
                    rderror("invalid char",
                            mkfix(v));
                c = v;
            }
            else if (0 == u) {
                u = c;
            }
        }
        name[i] = c;
        c = readc();
    }
    name[i] = 0;
    if (u) rderror("unknown slash sequence",
                   mkchar(u));
    if (i >= TOKLEN)
        rderror("string too long",
                mkstr(name, i));
    if (u) return NIL;
    n = mkstr(name, i);
    tag(n) |= CONST_TAG;
    return n;
}
```

The **rdquote()** function does not really read anything. It is used to create forms like **(quote x)**, **(unquote x)**, etc., where q is the desired quotation keyword.

Note that

```
cons(q, cons(n, NIL))
```

is safe, because q is an interned symbol and hence inherently GC-protected.

```
cell rdquote(cell q) {
    cell    n;

    Quoting++;
    n = xread2();
    Quoting--;
    return cons(q, cons(n, NIL));
}
```

The **meta()** function reads the *abbreviations* or *meta commands* listed in figure 24.

Abbreviation	Expands to
`,c` *text*	`(syscmd "`*text*`")`
`,h` *text*	`(help "`*text*`")`
`,l` *file*	`(load "`*file*`.ls9")`

Fig. 24 – Meta commands (abbreviations)

The abbreviations listed in the left column of the figure can be entered at the REPL (outside of lists or vectors), where they will expand to the corresponding expressions in the right column.

meta() returns the expanded expression.

```
cell meta(void) {
    int     c, cmd, i;
    cell    n, cmdsym;
    char    s[128];

    cmd = tolower(readc());
    c = readc();
    while (' ' == c) c = readc();
    i = 0;
    while (c != '\n' && c != EOF) {
```

```
        if (i < sizeof(s) - 6)
            s[i++] = c;
        c = readc();
    }
    rejectc(c);
    s[i] = 0;
    if ('l' == cmd) strcat(s, ".ls9");
    n = mkstr(s, strlen(s));
    n = 0 == i? NIL: cons(n, NIL);
    protect(n);
    switch (cmd) {
    case 'c':  cmdsym = symref("syscmd"); break;
    case 'h':  cmdsym = symref("help"); break;
    case 'l':  cmdsym = P_load; break;
    default:   prints(",c = syscmd"); nl();
               prints(",h = help"); nl();
               prints(",l = load"); nl();
               return NIL;
    }
    unprot(1);
    return cons(cmdsym, n);
}
```

The **comment()** function reads a comment. A comment begins with a semicolon (";") character and extends up to, and including, the end of the line in which it started:

```
; this is a comment
```

Unlike in most other LISP systems, though, a comment is not being skipped over by the parser, but expands to the expression

```
(-- " this is a comment")
```

which will later be ignored by the compiler. Because a comment is an expression, though, it may not appear anywhere in a program, but only in the following places:

- at the top level
- in **prog** forms (including **lambda**), between expressions

- between embedded **def** and **defun** forms

The disadvantage of this approach is that comments may not be placed anywhere, the advantage is that comments will be preserved when reading and then printing the forms in which they are contained. For instance, code formatters ("grinders", "pretty printers") will preserve LISP9 comments.

This approach was first realized in *MACLISP* [Moon1974], which used the **comment** keyword in the place of **--**. However, MACLISP also had semicolon comments, which were skipped by the reader.

Note that **read** will immediately return a comment after reading it (the asterisk is the LISP9 prompt):

```
* (read) ;comment
;comment
*
```

When a comment is entered at the REPL, compilation and evaluation will commence immediately, resulting in **NIL**, because an empty program is being run and the initial value of the result register is **NIL**:

```
* ;comment
nil
*
```

```
cell comment(void) {
    int     c, i;
    char    b[TOKLEN+1];

    i = 0;
    c = readc();
    while (c != '\n' && c != EOF) {
        if (TOKLEN == i)
            rderror("comment too long",
                    mkstr(b, i));
        if (i < TOKLEN) {
            b[i] = c;
            i++;
```

```
        }
        c = readc();
    }
    Line++;
    b[i] = 0;
    return cons(S_comment,
                    cons(mkstr(b, i), NIL));
}
```

The **xread2()** function is the main function of the reader. It skips over *white space characters* (ASCII HT, LF, CR, and blank) and then dispatches reading of specific forms to the functions discussed previously in this chapter. It reports various syntax errors, such as unexpected closing parentheses, *free dots* (dots that are not part of a dotted list), and unexpected input characters.

When no input is available on its input source, the function returns **EOFMARK**, which is a special object that serves as an *end of file* indicator.

Abbreviation	Expression
`'form`	`(quote form)`
`@form`	`(qquote form)`
`` `form ``	`(qquote form)`
`,form`	`(unquote form)`
`,@form`	`(splice form)`

Fig. 25 – Abbreviated quotation

xread2() expands the abbreviations listed in figure 25. Unlike the abbreviations accepted by **meta()**, these may appear anywhere in a program (but **,form** will not work at the top level, because it will be mistaken for a meta command).

xread2() will be called recursively by other functions of the reader. For instance, **rdlist()** will call this function to read individual elements of a list.

```
cell xread2(void) {
    int     c;
```

```
c = readc();
while (' ' == c ||
       '\t' == c ||
       '\n' == c ||
       '\r' == c)
{
    if (Intr || Readerr) return NIL;
    if ('\n' == c) Line++;
    c = readc();
}
if (Intr || Readerr) return NIL;
if (EOF == c) {
    return EOFMARK;
}
else if ('#' == c) {
    c = readc();
    if ('\\' == c) return rdchar();
    else if (LP == c) return rdvec();
    else if (isdigit(c)) return rdfix(c);
    else rderror("bad # syntax", mkchar(c));
}
else if ('"' == c) {
    return rdstr();
}
else if (LP == c) {
    return rdlist();
}
else if (RP == c) {
    if (!Inlist) rderror("unexpected ')'",
                         UNDEF);
    return RPAREN;
}
else if ('\'' == c) {
    return rdquote(S_quote);
}
else if ('`' == c || '@' == c) {
    return rdquote(S_qquote);
```

```
    }
    else if (',' == c) {
        if (!Inlist && !Quoting) return meta();
        c = readc();
        if ('@' == c) return rdquote(S_splice);
        rejectc(c);
        return rdquote(S_unquote);
    }
    else if (';' == c) {
        return comment();
    }
    else if ('.' == c) {
        if (!Inlist)
            rderror("unexpected '.'", UNDEF);
        return DOT;
    }
    else if (symbolic(c)) {
        return rdsymfix(c, 10, 1);
    }
    else {
        rderror("funny input character, code",
                mkfix(c));
    }
    return NIL;
}
```

xread() is the principal reader function. It resets the **Inlist** and **Quoting** variables and then delegates the real work to **xread2()**. If **Intr** is set when **xread2()** returns, **xread()** will signal an error. In this case, control will never return to the caller of **xread()**. This allows the user to abort partial input at the REPL.

When no errors occurred, **xread()** returns the internal form corresponding to the external form it has parsed.

```
cell xread(void) {
    cell    x;
```

```
    Inlist = 0;
    Quoting = 0;
    Readerr = NULL;
    x = xread2();
    if (Intr) error("aborted", UNDEF);
    return x;
}
```

13. The Printer

The *printer* is the counterpart to the reader. It translates the internal representations of LISP objects (“trees of nodes”) to a readable, *external representation*. In this case, “readable” means readable by **read** and not just readable by a human being.

Not all LISP objects do have an external representation, though, and when passed such an object, the printer will at least try to output something that makes sense to a human being, even if it is not readable in the sense of **read**.

If an object does have an external form, the printer will output its *canonical external form*. For instance, it will output

- fixnums as decimal numbers without a leading “**+**” sign and without any leading zeros
- strings with the minimal number of slashified characters
- unprintable chars using their symbolic char names, if available
- lists and vector with one blank between elements
- quotation in its abbreviated form, e.g.: **'x** instead of **(quote x)**
- comments in their abbreviated (semicolon) form

Objects that do not have an external representation include the *eof marker*, the *undefined value*, *functions*, *I/O ports*, and *catch tags*. When passed such an object, the printer will output some descriptive text, like

```
#<eof>  #<function>  #<inport 0>  #<catch tag>
```

Because the character “**<**” will not be accepted by **read** after a “**#**” character, these sequences of characters are *unreadable* in the sense of **read**.

The printer also has a *pretty-printing* mode, where it will emit “beautiful” output instead of external representations. For instance, in default (non-pretty) print mode, it will emit **#\A** for the char object holding the code point of “**A**”, but in pretty-print mode, it will just emit the character “**A**”. Details about pretty-printing will

follow later in this chapter.

The printer is available to LISP programs by means of multiple functions, the principal ones being **prin**, which outputs external representations, and **princ**, which pretty-prints objects.

The **ntoa()** function writes the external representation of the fixnum x_r (x to the base r) to an internal buffer and returns a pointer into that buffer. The assertion $2 \le r \le 36$ must hold. The size of the buffer must be increased, if your **int** type should be more than 198 bits wide.

Digits with a value greater than 10 will be represented by lower case letters.

```
char *ntoa(int x, int r) {
    char    buf[200];
    int     i = 0, neg;
    char    *p = &buf[sizeof(buf)-1];
    char    d[] = "0123456789abcdefgh"
                  "ijklmnopqrstuvwxyz";

    neg = x<0;
    *p = 0;
    while (x || 0 == i) {
        i++;
        p--;
        *p = d[abs(x % r)];
        x = x / r;
    }
    if (neg) {
        p--;
        *p = '-';
    }
    return p;
}
```

The **prchar()** function prints the external representation of the *char object* x. It will print **#\c** for printable ASCII characters **c**, **#\ht** for

horizontal tab (HT) characters, **#\nl** for newline characters, **#\sp** for blanks, and an octal sequence beginning with **#** for all other non-printable characters.

When the *sl* (slashify) argument is zero (indicating pretty-print mode), the character represented by the char object x is written to the current output port.

For example, printing **#\nl** using $sl = 1$ will print **#\nl**, but printing it with $sl = 0$ will advance the cursor to the beginning of the next line (if the output port is connected to a terminal).

```
void prchar(int sl, cell x) {
    if (sl) {
        prints("#\\");
        if (9 == charval(x))
            prints("ht");
        else if (10 == charval(x))
            prints("nl");
        else if (' ' == charval(x))
            prints("sp");
        else if (charval(x) < 32 ||
                 charval(x) > 126)
        {
            prints("\\");
            prints(ntoa(fixval(x), 8));
        }
        else writec(charval(x));
    }
    else {
        writec(charval(x));
    }
}
```

The **prfix()** function prints the decimal representation of the numeric value of the *fixnum object* x.

```
void prfix(cell x) {
```

```
    prints(ntoa(fixval(x), 10));
}
```

The **prstr()** function prints the external representation of a *string object*, translating characters in a similar way as **prchar()**, but HT becomes "**\t**" and newline becomes "**\n**". All other non-printable characters will be replaced by the shortest possible octal sequence. I.e. the natural length of the octal number will be used unless the following character is also an octal digit. In the latter case, a three-digit octal sequence will be used. For instance:

```
"a\2b"  "1\0023"
```

In the left example, the ASCII STX character (code point 2) is is represented by "**\2**", because the following character, "**b**" is not an octal digit. In the right example, "**\002**" is used to represent STX in order to separate its octal code from the subsequent "**3**".

When $sl = 0$, **prstr()** will pretty-print the string: it will omit the enclosing double quote characters and emit each character as is, without ever slashifying it.

```
void prstr(int sl, cell x) {
    int     i, c;

    if (sl) {
        writec('"');
        for (i=0; i<stringlen(x)-1; i++) {
            c = (byte) string(x)[i];
            if ('"' == c)
                prints("\\\"");
            else if ('\\' == c)
                prints("\\\\");
            else if (10 == c)
                prints("\\n");
            else if (c < ' ' || c > 126) {
                writec('\\');
                if (octalp(string(x)[i+1])) {
                    if (c < 100) writec('0');
```

```
                        if (c < 10) writec('0');
                    }
                    prints(ntoa(c, 8));
                }
                else
                    writec(c);
            }
            writec('"');
        }
        else {
            printb(string(x));
        }
}
```

The **prlist()** function prints the *list*, *dotted list*, or *pair* x. When printing a list, its individual elements will be separated by single blanks. No blank character will print after the opening parenthesis or in front of the closing parenthesis. When the list is a dotted list (or a pair), a blank, a dot ("."), and another blank will separate the last from the second-to-last element.

The sl (slashify) parameter is not used by **prlist()** itself, but will be passed to **prex()**, which prints the individual members of a list.

The d ("depth") parameter keeps track of the depth (the number of nested car parts) of the list being printed. This is done in order to detect (potentially) cyclic structures, which will abort printing.

```
void prex(int sl, cell x, int d);

void prlist(int sl, cell x, int d) {
    writec(LP);
    while (x != NIL && Plimit != 1) {
        prex(sl, car(x), d+1);
        x = cdr(x);
        if (x != NIL) {
            writec(' ');
            if (atomp(x)) {
                prints(". ");
```

```
                        prex(sl, x, d+1);
                        break;
                  }
            }
      }
      writec(RP);
}
```

The **prvec()** function prints a LISP9 *vector*. Like the function printing lists, it will place a single blank between individual elements of a vector, but not before the first one or after the last one.

The *sl* and *d* parameter serve the same purposes as in **prlist()**.

```
void prvec(int sl, cell x, int d) {
      int     i;

      prints("#(");
      for (i=0; i<veclen(x); i++) {
            prex(sl, vector(x)[i], d+1);
            if (i < veclen(x)-1) writec(' ');
      }
      writec(')');
}
```

The **prport()** function prints some unreadable (by **read**) informative text describing the *port object* *x*, e.g.:

#<inport *n*>
#<outport *n*>

The number *n* following the word **inport** or **outport** indicates the *port number* stored in the object, i.e. the port to which it is connected.

The *out* argument indicates an outport when non-zero and in input port otherwise.

```
void prport(int out, cell x) {
    prints("#<");
    prints(out? "out": "in");
    prints("port ");
    prints(ntoa(portno(x), 10));
    prints(">");
}
```

The **pruspec()** ("print unknown special") and **pruatom()** ("print unknown atom") functions are last-resort functions that will print some informative text when the printer is fed an object that it does not know how to print. **pruspec()** will be used for unknown *special objects* and **pruatom()** for unknown *atoms* (or *nodes*). **pruspec()** will print

#<special object *n*>

and **pruatom()** will print

#<atom *n*>

where *n* is the value of a special object itself or the value stored in the car field of an atom. In normal operation, the user should never see any of the above sequences. The latter can be provoked, though, by *untagging* a fixnum or a char (see the **untag** function for details).

```
void pruspec(cell x) {
    prints("#<special object ");
    prints(ntoa(x, 10));
    prints(">");
}

void pruatom(cell x) {
    prints("#<atom ");
    prints(ntoa(car(x), 10));
    prints(">");
}
```

The **quoted()** macro returns truth, if its argument *x* is a two-

element list with the symbol *q* as its car part. It is used to identify forms that are used in *quotation*, i.e. applications of **quote**, **qquote**, **unquote**, and **splice**.

The **prquote()** function prints any of the above forms as an abbreviation (e.g. **@x** instead of **(qquote x)**), where *x* is the form to print and *sl* and *d* are the usual slashification and depth parameters.

```
#define quoted(x, q) \
    (car(x) == (q) && \
     cdr(x) != NIL && \
     NIL == cddr(x))

void prquote(int sl, cell x, int d) {
    if (car(x) == S_quote) writec('\'');
    else if (car(x) == S_qquote) writec('@');
    else if (car(x) == S_unquote) writec(',');
    else if (car(x) == S_splice) prints(",@");
    prex(sl, cadr(x), d);
}
```

The **commentp()** macro returns truth, if the form *x* is a *comment*, i.e. an application of **--**.

prcomm() prints a comment in its abbreviated form, which is

```
;comment
```

instead of

```
(-- "comment")
```

When the **--** keyword has multiple arguments, it prints all of them, separated by blanks. All arguments of the comment function will print in their unslashified form. E.g.

```
(-- "foo" "\"bar\"" #\X)
```

will print as

```
;foo "bar" X
```

and the above will read as

```
(-- "foo \"bar\" X")
```

Hence it is a good idea to always use the abbreviated form of comments and *if* the expanded form is used, there should be a single string argument containing the comment. Using the abbreviated form has the advantage of taking care of slashification of characters that need slashification.

```
#define commentp(x) (car(x) == S_comment)

void prcomm(cell x, int d) {
    writec(';');
    x = cdr(x);
    while (x != NIL) {
        prex(0, car(x), d);
        x = cdr(x);
        if (x != NIL) writec(' ');
    }
}
```

The **prex()** ("print expression") function dispatches printing of any object x to the functions listed above. It handles only a few very simple cases, like **#<function>** or **#<eof>**, itself.

When the *depth* (d) of the object to print exceeds **PRDEPTH**, an error will be signalled, assuming that the object cycles through a car field of a pair or through a vector slot.

When the **Intr** flag becomes active while printing, the **prex()** function will also signal an error, thereby allowing to interrupt the printing of objects whose external representations are very long or cycle through their cdr fields.

```
void prex(int sl, cell x, int d) {
    if (d > PRDEPTH) {
        prints("\n");
        error("prin: nesting too deep", UNDEF);
    }
    if (Intr) {
```

```
        Intr = 0;
        error("interrupted", UNDEF);
    }
    if (NIL == x) prints("nil");
    else if (TRUE == x) prints("t");
    else if (EOFMARK == x) prints("#<eof>");
    else if (UNDEF == x) prints("#<undef>");
    else if (charp(x)) prchar(sl, x);
    else if (fixp(x)) prfix(x);
    else if (symbolp(x)) printb(symname(x));
    else if (stringp(x)) prstr(sl, x);
    else if (vectorp(x)) prvec(sl, x, d);
    else if (closurep(x)) prints("#<function>");
    else if (ctagp(x)) prints("#<catch tag>");
    else if (inportp(x)) prport(0, x);
    else if (outportp(x)) prport(1, x);
    else if (specialp(x)) pruspec(x);
    else if (atomp(x)) pruatom(x);
    else if (quoted(x, S_quote))
        prquote(sl, x, d);
    else if (quoted(x, S_qquote))
        prquote(sl, x, d);
    else if (quoted(x, S_unquote))
        prquote(sl, x, d);
    else if (quoted(x, S_splice))
        prquote(sl, x, d);
    else if (commentp(x)) prcomm(x, d);
    else prlist(sl, x, d);
}
```

The **xprint()** function is the primary printer interface. It prints the object x in **read**-able form, if $sl = 1$ and pretty-prints it, if $sl = 0$. The sl argument indicates *slashification*, a concept that dates back to LISP versions that predate even MACLISP. It probably appeared first in PDP-6 LISP [AIM116A]. PDP-1 LISP [DB1964], which later became BBN LISP, was probably the first LISP system that used a backslash instead of a forward slash to slashify special characters.

Slashification was originally used to “escape” reserved characters

in symbol names, like

```
foo/ bar   ; (symbol "foo bar")
a//b       ; a/b
/(x/)      ; (symbol "(x)")
/          ; (symbol " ")
```

This was necessary, because symbols were sometimes used in the place of strings. Modern LISP dialects typically use the backslash character instead of a slash and extend the principle to strings. LISP9 does not support slashification in symbols, but does so in strings.

The slashify parameter in LISP9's print, however, does not only activate translations like *newline* →**\n**, but also makes the printer emit external representations instead of values. For example, printing **#\x** with slashification will output the char object **#\x**, but *pretty-printing* it (without slashification) will print its value, **x**.

The original meaning of the term "slashification" is narrower than used in this text. Apologies to those who know better!

When the printer hits a *printer limit* during output (indicated by the **Plimit** variable becoming equal to 1), the **xprint()** function appends an ellipsis ("**...**") to its output to inform the user that output its is incomplete. This feature is used by the **report()** function to print erroneous parts of expressions.

```
void xprint(int sl, cell x) {
    prex(sl, x, 0);
    if (1 == Plimit) {
        Plimit = 0;
        prints("...");
    }
}
```

The **prin()**, **princ()**, and **print()** functions resemble the general purpose output functions of LISP9 (although the **print** function of LISP9 is variadic).

prin() prints its argument with slashification, **princ()** pretty-prints

its argument, and **print()** prints its argument like **prin()**, but in addition emits a newline sequence after its output.

```
void prin(cell x) { xprint(1, x); }

void princ(cell x) { xprint(0, x); }

void print(cell x) { prin(x); nl(); }
```

14. The Compiler

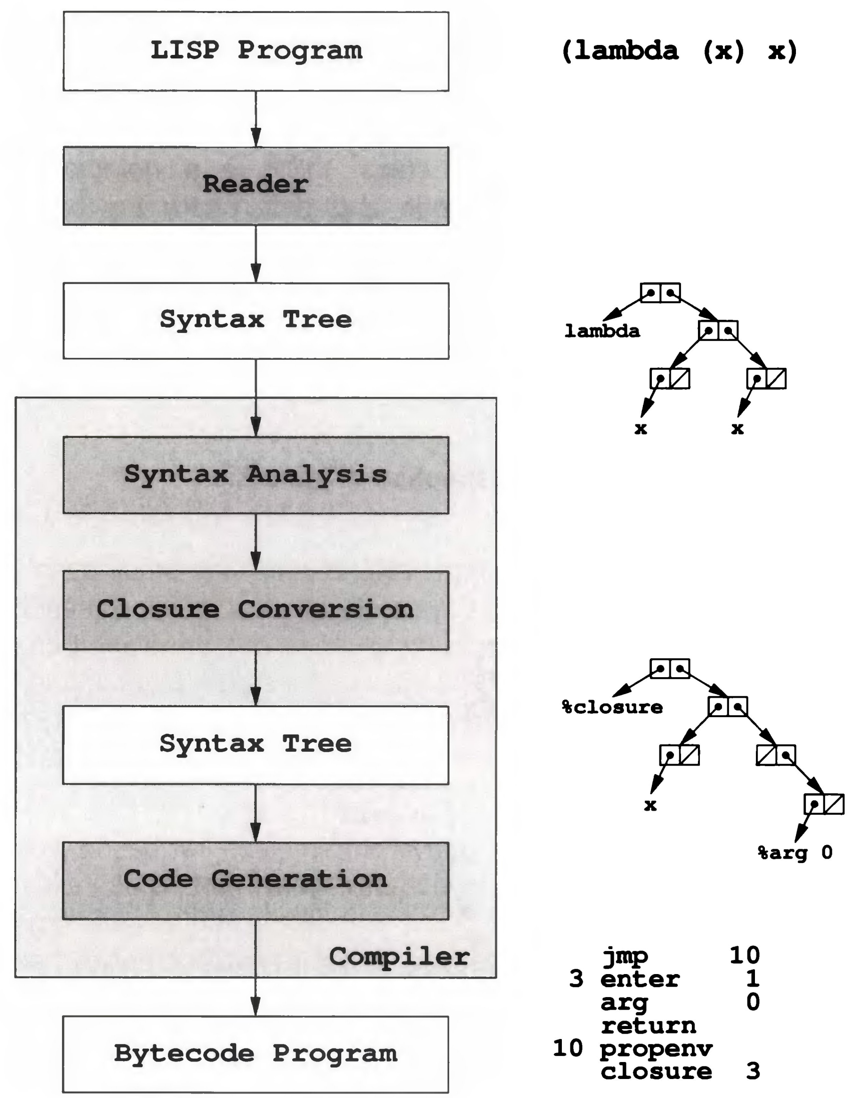


Fig. 26 – The LISP9 compiler

This chapter describes the *compiler* of the LISP9 system. The compiler is the part of the system that translates the internal representation of LISP programs (as produced by the *reader*) to

abstract machine code that will be interpreted by an abstract machine in the last step of the process of evaluation.

Figure 26 illustrates the process of compilation with a textual representation of LISP programs as its input and bytecode for an abstract machine as its output.

The first step in the process is to translate the textual representation of the input program to a structure that is easier to manipulate in the subsequent steps. LISP is a *homoiconic* language, which means that code and data share the same representation. When a LISP expression is read by the **read** function, the *external* (textual) representation of the expression is translated to some LISP data object – typically a nested *list*.

In figure 26, the example program

```
(lambda (x) x)
```

is translated to the nested list structure in figure 27.

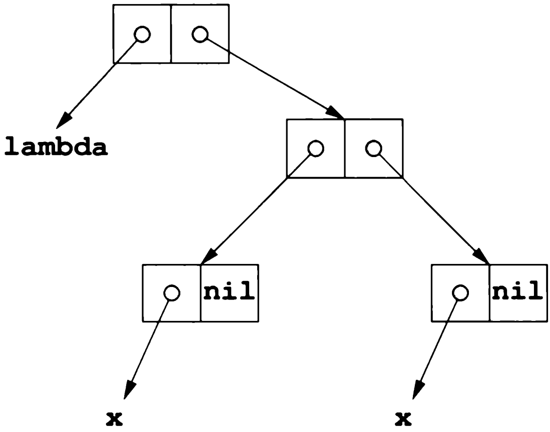


Fig. 27 – List structure as abstract syntax tree

Compilers typically operate on structures called *abstract syntax trees* (*AST*). Such structures represent the input program in tree form, where inner nodes indicate operations and outer nodes denote operands. The part of the process that converts a textual program to an AST is called a *parser*. Because LISP is homoiconic, the **read** function can serve as a parser for LISP.

In most LISP syntax trees, inner nodes will denote either

application or concatenation. For instance, figure 28 shows the tree form of the expression **(cons foo bar)**, which applies **cons** to the concatenation of **foo**, **bar**, and **nil**. However, some trees, like the tree in figure 27, which was generated from the lambda expression **(lambda (x) x)**, have special meanings. For example, the **(x)** in the expression is *not* a function application.

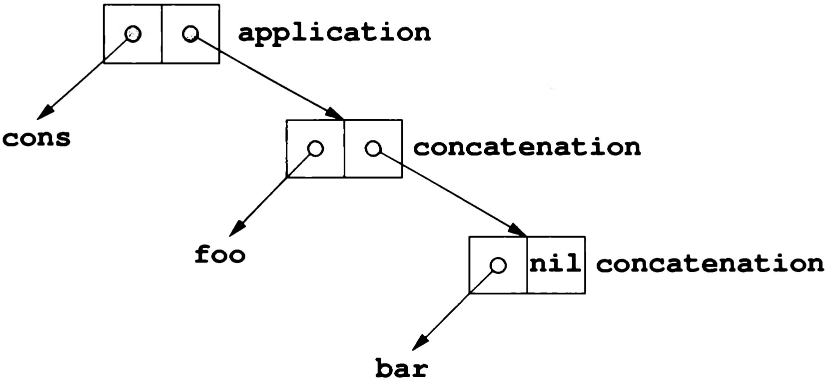


Fig. 28 – Application and concatenation

Hence some additional analysis has to be performed. The reader does not "know" about LISP's syntax. It treats all its input as data and transforms it to trees (or other objects), no matter whether or not that input represents a valid LISP program. For example, it will happily turn expressions like **(lambda)** or **(quote foo bar baz)** or **(setq 5 7)** into corresponding list structures.

Hence the first phase of the compiler itself will be *syntax analysis*. This phase will examine the syntax trees generated by **read** and make sure that they represent syntactically correct LISP9 programs. When an error is found at this stage, compilation will be aborted and control will be passed back to the invoking process (either the REPL or, in case of batch processing, the operating system).

14.1 Syntax Analysis

One of the most obvious syntax tests is checking the number of arguments given to a particular keyword. For example,

(setq *x y***)**

might be correct, but at least x has to be inspected more closely. The expressions

(setq *x***)**
(setq *x y z***)**

though, do not need to be tested any further, because **setq** expects exactly two arguments.

The **length()** function returns the length of the proper (non-dotted) list n. It will be used to count arguments.

```
int length(cell n) {
    int     k;

    for (k = 0; n != NIL; n = cdr(n))
        k++;
    return k;
}
```

The **ckargs()** function makes sure that the special form x has at least *min* and at most *max* arguments. When $max < 0$, then there is no upper limit on the arguments in x. **ckargs()** signals an error when the wrong number of arguments is found in x.

```
void ckargs(cell x, int min, int max)
{
    int     k;
    char    buf[100];

    k = length(x)-1;
    if (k < min || (k > max && max >= 0)) {
        sprintf(buf,
          "%s: wrong number of arguments",
          symname(car(x)));
        error(buf, x);
    }
}
```

The **ckseq()** function performs syntax analysis on each element of the list x. The *top* argument indicates that syntax checking is currently being applied to a form that appears at the top level of a program, i.e. a form that does not appear inside of another form.

The *comm* argument indicates that a *comment* may appear in the place of each of the subforms of x.

All syntax analysis functions return 0, but that value does not bear any significance. See the **syncheck()** function at the end of this section for an explanation.

```
int syncheck(cell x, int top, int comm);

int ckseq(cell x, int top, int comm) {
    for (; pairp(x); x = cdr(x))
        syncheck(car(x), top, comm);
    return 0;
}
```

The **ckapply()** function makes sure that **apply** has at least two arguments. That's all. Not much can be done in a dynamic programming language at this point without resorting to more complex static program analysis, like type inference. [Milner1978]

```
int ckapply(cell x) {
    ckargs(x, 2, -1);
    return 0;
}
```

A comment may only appear in a context where comments are allowed. This is what the **ckcomment()** function verifies.

```
int ckcomment(int comm) {
    if (!comm) error("comment in wrong context",
                     UNDEF);
    return 0;
```

```
}
```

A *definition* is always introduced by a **def** form in LISP9. All derived forms, like **defun**, will be transformed to applications of **def**. The **ckdef()** function makes sure that the first argument of **def** is a symbol and the second one is syntactically correct.

Note that *top=0* and *comm=0* are passed to **syncheck()**, because the *value* in **(def** *symbol value***)** is not at the top level (it is contained in **(def ...)**) and it may not be a comment. The value may *contain* comments, though, for instance when it is a **lambda** special form).

```
int ckdef(cell x, int top) {
    ckargs(x, 2, 2);
    if (!symbolp(cadr(x)))
        error("def: expected symbol", cadr(x));
    if (!top)
        error("def: must be at top level", x);
    return syncheck(caddr(x), 0, 0);
}
```

The **if** form has two or three arguments and the **if*** form has exactly two arguments. This is what the **ckif()** and **ckifstar()** functions test. They also make sure that those arguments are syntactically correct.

```
int ckif(cell x) {
    ckargs(x, 2, 3);
    return ckseq(cdr(x), 0, 0);
}

int ckifstar(cell x) {
    ckargs(x, 2, 2);
    return ckseq(cdr(x), 0, 0);
}
```

The **symlistp()** predicate returns truth, if its argument x is either a symbol or a list of symbols. If it is a list, the list may even be dotted or empty. When it is dotted, the object at the end of the list must also be a symbol.

This function is used to make sure that the *formal argument lists* of **lambda** are in the proper shape.

```
int symlistp(cell x) {
    cell    p;

    for (p = x; pairp(p); p = cdr(p)) {
        if (!symbolp(car(p)))
            return 0;
    }
    return symbolp(p) || NIL == p;
}
```

The **memq()** function (almost) mimics Scheme's **memq** function [R4RS]. It returns the first tail of x whose first element is **eq** to a. If if x does not contain a, it return **NIL**. The function assumes that a is a proper (NIL-terminated) list.

```
int memq(cell x, cell a) {
    for (; a != NIL; a = cdr(a))
        if (car(a) == x) return a;
    return NIL;
}
```

The **uniqlistp()** predicate returns truth, if no two elements of the (proper) list x are equal in the sense of **eq**. It is used to make sure that formal argument lists of **lambda** do not contain any duplicate symbols.

```
int uniqlistp(cell x) {
    if (NIL == x) return 1;
    while (cdr(x) != NIL) {
```

```
        if (memq(car(x), cdr(x)) != NIL)
            return 0;
        x = cdr(x);
    }
    return 1;
}
```

The **flatargs()** function turns the (potentially dotted) list a into a flat (proper) list. If a is a symbol, it returns a singular list containing that symbol. It is used to "flatten" formal argument list of **lambda**, e.g.:

```
(a b . c)  →  (a b c)
a          →  (a)
```

```
cell flatargs(cell a) {
    cell    n;

    protect(n = NIL);
    while (pairp(a)) {
        n = cons(car(a), n);
        car(Protected) = n;
        a = cdr(a);
    }
    if (a != NIL) n = cons(a, n);
    unprot(1);
    return nreverse(n);
}
```

The **cklambda()** function makes sure that the lambda form x

- has at least two arguments
- has a proper formal argument list as its second argument
- has no duplicate formal arguments
- has a syntactically correct body (2nd and following arguments)

Note that comments are allowed in the body of **lambda**. In fact, the *only* expression in the body of a lambda function might be a

comment, which would make the value of the function undefined. A more sophisticated syntax analyzer might catch this case. Feel free to implement the necessary code.

```
int cklambda(cell x) {
    ckargs(x, 2, -1);
    if (!symlistp(cadr(x)))
        error("lambda: invalid formals",
              cadr(x));
    if (!uniqlistp(flatargs(cadr(x))))
        error("lambda: duplicate formal",
              cadr(x));
    return ckseq(cddr(x), 0, 1);
}
```

A **macro** definition has basically the same shape as a **def** form, only with a different keyword as its first element. The **ckmacro()** function syntax-checks this form. The derived **defmac** form will be transformed to an application of **macro** during macro expansion.

```
int ckmacro(cell x, int top) {
    ckargs(x, 2, 2);
    if (!symbolp(cadr(x)))
        error("macro: expected symbol", cadr(x));
    if (!top)
        error("macro: must be at top level", x);
    return syncheck(caddr(x), 0, 0);
}
```

A **prog** form is correct, if all of its arguments are correct. **ckprog()** performs this test.

```
int ckprog(cell x, int top) {
    return ckseq(cdr(x), top, 1);
}
```

The **ckquote()** function makes sure that **quote** has exactly one argument.

```
int ckquote(cell x) {
    ckargs(x, 1, 1);
    return 0;
}
```

As indicated at the beginning of this section, **setq** has exactly two arguments and, given this case, the first one has to be a symbol. The **cksetq()** function performs the syntax analysis of **setq**.

```
int cksetq(cell x) {
    ckargs(x, 2, 2);
    if (!symbolp(cadr(x)))
        error("setq: expected symbol", cadr(x));
    return ckseq(cddr(x), 0, 0);
}
```

The **syncheck()** function performs syntax analysis of the form x. When $top \neq 0$, the form x appears at the top level and when $comm \neq 0$, the form may be a comment. **syncheck()** is called recursively by the above functions, passing the proper values for top and $comm$ along.

Any *atom* is a valid LISP9 program, so **syncheck()** returns immediately when given an atom. Dotted lists are never valid programs. All other cases will be delegated to the functions above. The default case is *function application*, which is simply checked for correctness of its function and arguments.

syncheck() returns an uninteresting value when x is a valid form. Otherwise it signals an error.

The reason why the syntax analyzing functions return a value at all is the structure of the **syncheck()** function. Because all the **ck...()** functions return a value, delegation to other functions can be done by statements like

```
if (car(x) == S_apply) return ckapply(x);
```

which looks less cumbersome than

```
if (car(x) == S_apply) { ckapply(x); return; }
```

Yes, equally readable alternatives (like nested "if") exist.

```
int syncheck(cell x, int top, int comm) {
    cell    p;

    if (atomp(x)) return 0;
    for (p = x; pairp(p); p = cdr(p))
        ;
    if (p != NIL)
        error("dotted list in program", x);
    if (car(x) == S_apply) return ckapply(x);
    if (car(x) == S_comment)
        return ckcomment(comm);
    if (car(x) == S_def) return ckdef(x, top);
    if (car(x) == S_if) return ckif(x);
    if (car(x) == S_ifstar) return ckifstar(x);
    if (car(x) == S_lambda) return cklambda(x);
    if (car(x) == S_macro)
        return ckmacro(x, top);
    if (car(x) == S_prog) return ckprog(x, top);
    if (car(x) == S_quote) return ckquote(x);
    if (car(x) == S_setq) return cksetq(x);
    return ckseq(x, top, comm);
}
```

14.2 Closure Conversion

Once a form is known to be in the proper shape, its handling becomes much simpler, because lots of assumptions about its syntax tree can be made. For instance, the body of a lambda function f can now be referred to as $cddr(f)$ without having to make sure that the form f does indeed have a cddr field. Since the form has passed syntax analysis, it does. Similarly, it can be assumed safely that the first arguments of the **def**, **macro**, and

setq forms are symbols.

The next step in the compilation of LISP code is called *closure conversion*. A *closure* is a technical term that describes what LISP programmers typically call a *function* – given *lexical scoping*.

When a LISP function is created, it *closes over* all free variables referenced in its body. The *free variables* of a function are those variables that are referenced in a function, but are are not formal arguments of the same function. A variable that is not free in a function is called *bound* in that function (and vice versa).

For instance, the variable x is bound in

```
(lambda (x) (f x))
```

but the variable f is free in that expression – because it is not contained in the formal argument list **(x)**.

The function created by the lambda expression will "remember" the binding of f, though, thereby enabling definitions like this:

```
(defun (complement p)
  (lambda (x) (not (p x))))
```

The **complement** function takes a predicate p as its argument and creates a new (anonymous) function that returns the complement of p applied to x (its own argument). When **complement** is done evaluating, the variable p in **complement** no longer exists, but the resulting anonymous function will remember its binding. This is why the following code works:

```
(defun (even x) (= 0 (rem x 2)))
(def odd (complement even))
(odd 5)  =>  t
```

LISP textbooks, including this one, often talk about *bindings* of variables to values or of symbols to values. This terminology is a bit muddy, though, so it shall be clarified in the following.

A *variable* is a *symbol* that is bound to a *location*. A location is a storage cell – sometimes called a *box* – in which some value can be stored. So when a "variable is bound to a value", this actually means that the "variable is bound to a location containing the value".

The term “the binding of a variable” is sometimes used to refer to the box that holds the value of that variable. The binding of a variable is “in effect”, or exists, as long as the box associated with the name of the variable exists.

Bindings of symbols to locations are created by **def** forms or by lambda functions. All bindings have *indefinite extent*: once created, they exist forever. However, *local bindings*, as created by **lambda**, may be recycled by the garbage collector when it can prove that these bindings can no longer be accessed by any program in the system.

In programming languages without closures, a function is merely an address in memory. Application of the function is performed by temporarily transferring control to that address. It works the same way in LISP, but in addition the bindings that were in effect when the function was created also have to be in effect when the function is being applied.

The task of closure conversion is basically to replace all references to free variables in a function by references to a *local environment* that will become part of the data object representing a function. In LISP, this conversion can be though of as:

```
(lambda (z)  -->  (%closure (z)
  (z x y))          #(<binding of x>
                      <binding of y>)
                    (z (%ref 0) (%ref 1)))
```

The **%closure** keyword indicates a closure; x has been replaced by **(%ref 0)** and y by **(%ref 1)** in the body of the closure. In front of the body, there is another element in the closure: a vector containing the local environment. Given this vector v, **(%ref** n**)** essentially becomes **(vref** v n**)**.

In the above expression the free variables x and y have been replaced by references to locations in a local environment. When the closure is applied to a value, the local environment becomes the current environment for the duration of evaluating the body of the closure.

There is one problem, though: The bindings of x and y cannot

necessarily be known at compile time, so the compiler cannot build the vector implementing the local environment. This has to be done *at run time*.

14.2.1 Building an Environment

Every free variable is bound *somewhere*, either in the environment of an *outer function* or in the formal argument list of the *immediate* outer function (*IOF*). An outer function of the function f is any function in which f is contained. Given $f = lambda\ (z)\ (z\ x\ y))$ and

```
(lambda (x) (lambda (y) (lambda (z) (z x y))))
```

both the entire above expression as well as

```
(lambda (y) (lambda (z) (z x y)))
```

are outer functions of f. However, only the latter is the *immediate* outer function of f, because it directly contains f without any other functions (lambda expressions) in between.

In the example, the function f contains one free variable (y) that is bound in its immediate outer function and another one (x) that is bound in the outermost function. Note that x is also free in the IOF of f.

So when $f = (lambda\ (z)\ (z\ x\ y))$ is created, its environment has to be populated with one free variable of its IOF and one bound variable of its IOF. The former can be achieved by copying the binding of x from an environment slot of the IOF to a slot in the environment of the new function.

However, the same cannot be done with a variable that is *bound* in the IOF, because it is located on the runtime stack and does not have any environment slot. So the another step is necessary to replace bound variables in closures as well:

```
(lambda (z)   -->   (%closure (z)
  (z x y))            #(<binding of x>
                        <binding of y>)
                      ((%arg 0)   ; z
                       (%ref 0)   ; x
                       (%ref 1))) ; y
```

An expression of the form **(%arg** *n***)** now references the nth argument of the closure. The formal argument list is still kept, through, for two reasons: it indicates the number of arguments that the closure expects and it indicates whether the number of arguments is fixed or variable.

Now that all variables can be identified using numeric indices, the compiler can build the *initialization map* or *initmap* of the closure. This map specifies a source slot for each binding of a free variable as well as its destination slot in the new local environment. It also specifies the type of slot from which the binding will be fetched: either from an outer environment (e) of from an argument slot (a) on the runtime stack.

Given that

- the binding of x is in environment slot 0 of the IOF
- the binding of y is in argument slot 0 of the IOF

the initmap would look like this:

```
((e 0 0)    ; env slot 0 --> slot 0
 (a 0 1))   ; arg slot 0 --> slot 1
```

meaning that when (*lambda* (z) (z x y)) is being created, the first slot of its local environment is populated from slot 0 of the environment of the IOF and slot 1 is populated from argument 0 of the IOF. The complete conversion would then look like this:

```
(lambda (z)  -->  (%closure (z)
  (z x y))          ((e 0 0) (a 0 1))
                    ((%arg 0) (%ref 0)
                              (%ref 1)))
```

The body of the closure is now completely free of symbols and all references to values of variables are done via *shallow binding*, i.e. by directly accessing a slot of an environment or a slot of the runtime stack. Therefore, variable access is constant-time. The local environment itself has been replaced by instructions in the initmap that specify how to create the environment at run time.

Because there are no symbolic references in the body of the closure any longer, it can be translated to bytecode for an abstract

machine. The closure can then be applied to some values by temporarily making its local environment the current environment and transferring control to the address of its bytecode.

Note that local environments are *always* being populated from the IOF and *never* from any other outer functions. This can be demonstrated as follows:

If a variable x is bound in a function f, it shadows any variables named x that are bound in outer functions of f, so x is never fetched from *any* outer function.

A variable that is free in f is either bound or free in the IOF of f. In either case, its binding will be fetched from the IOF.

The binding of a variable that is free in both the IOF and the IOF of the IOF already has been *propagated* from the IOF of the IOF to the IOF. Hence there is no need to refer to any function other than the IOF when building a new environment.

Alternatively: given a function c with IOF b and outermost function a, there cannot be any variable that exists (free or bound) in a, does *not* exist in b and is free in c. It must also be free (or bound) in b and therefore its binding will be fetched from b.

When there is no outer function from which a binding can be copied, then the binding will be copied from the *top level environment* in which all bindings created by **def** are located. For instance:

```
(lambda (x)     -->   (%closure (x)
  (foo x x))            ((e 6 0))
                        ((%ref 0) (%arg 0)
                                  (%arg 0)))
```

given that the symbol **foo** is bound to slot 6 of the top level environment.

When bindings are propagated from one environment to the next, they do not necessarily occupy the same slots in every environment. Each time a new environment is built, bindings may vanish or be added. Bindings are added when a variable that was previously bound becomes free, as in

```
(lambda (x) (lambda (y) (x y)))
```

Here the variable x is bound in the outer, but free in the inner function. Bindings vanish when a free variable exists in the outer function, but not in the inner, as in

```
(lambda () (foo (lambda (y) y)))
```

Here the variable *foo* is free in the outer function, but does not exist in the inner one. Consider the expression

```
(lambda (f g)              ; E1
  (lambda (h)              ; E2
    (f (lambda (x)         ; E3
         (foo (g x) (h x))))))
```

where various free variables appear and disappear as the number of outer functions increases. The corresponding local environments, argument slots, and initmaps are shown in figure 29. E1 is the outermost environment, E2 the middle one, and E3 the innermost one. Solid arrows denote propagation of bindings, dashed arrows indicate time.

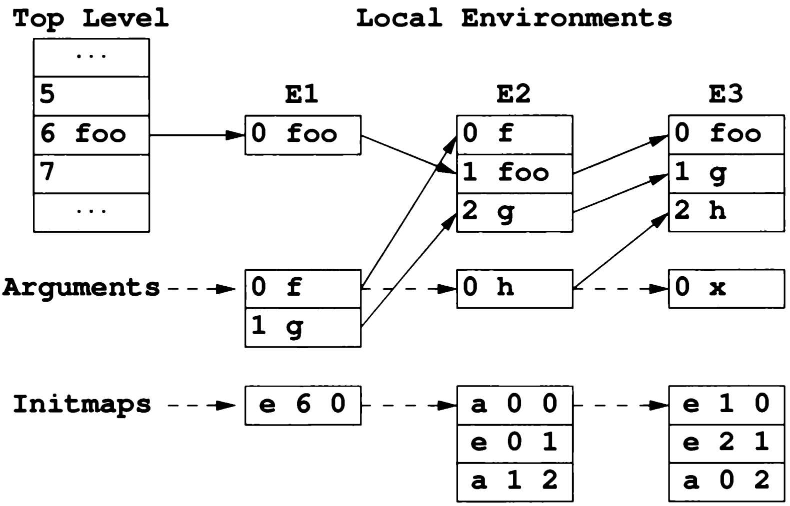


Fig. 29 – Propagation of bindings

Note that, although there are four free variables in total, no environment contains more than three bindings. This is because the binding of f is dropped at the same time as the binding of h is

added. Local environments do not carry any dead weight. Only variables that are actually free in a function will be propagated.

A special case of the closure is the *combinator*, a function without any free variables, like

```
(lambda (x) x)
```

The closure resulting from a combinator has an empty initmap, so no local environment has to be created at all. Strictly speaking, the resulting closure is not even a closure, because it does not close over any free variables. Combinators are very efficient and under some circumstances, closures can be converted to combinators.

The following closure conversion code will do all of the above and a few more things. It is the heart of the compiler, where most of the complexity resides. But let us approach this part step by step...

14.2.2 Finding Free Variables

The **set_union()** function returns the *union* of the sets a and b. Each set is represented by a list of unique symbols. The function returns a fresh list that contains every element that is contained in either a or b or both. Since it operates on lists of symbols and the sets a and b are typically small, it uses the **memq** function (rather than a hash table) to test whether a given element is already contained in the result.

```
cell set_union(cell a, cell b) {
    cell    n;

    a = reverse(a);
    protect(a);
    protect(n = b);
    while (pairp(a)) {
        if (memq(car(a), b) == NIL)
            n = cons(car(a), n);
        car(Protected) = n;
        a = cdr(a);
    }
    if (a != NIL && memq(a, b) == NIL)
```

```
        n = cons(a, n);
    unprot(2);
    return n;
}
```

The **freevars()** function returns a list of *free variables* in the *S-expression* x given the bound variables listed in the *environment* e.

The term "S-expression" ("symbolic expression") is an ancient umbrella term for all LISP expressions and forms, including special forms [McCarthy1960a]. Originally, the term applied only to symbols and pairs, but these days it is more broadly applied to all kinds of LISP data objects. The latter, more liberal usage will be adopted here.

The term "environment" is used rather loosely in the context of the **freevars()** function, because it is really only a list of bound symbols here and does not contain any bindings.

The function uses the **subrp()** predicate, which returns truth, when the symbol x names a *primitive* function. The terms shall be explained later. For now it is only important to know that the names of primitive functions are not considered to be variables when they appear in operator positions in function applications. Hence they cannot be free variables in this case.

The name *SUBR* is short for "subroutine", another ancient name for functions that were not implemented in LISP, but in assembly language or, later, in various system programming languages [MaCarthy1960b].

Basically, finding free variables is simple [Holm2016], but in the specific case discussed here, there are many different cases to consider.

(1) If x is in e, it is bound; the result is **NIL**.

(2) If x is a symbol (and not in e), it is free; the result is (x), a singular list containing x.

(3) If x is an atom, but not a symbol, it is not a variable at all; the result is **NIL**.

(4) If x is an application of **quote**, the result is **NIL**. Quoted objects can never be free variables.

(5) If x is an application of **--** (a comment), the result is **NIL**.

(6) If x is an application of **apply**, **prog**, **if**, **if***, **setq**, or an application of any primitive function, the result is the union of the free variables of $cdr(x)$.

(7) If x is an application of **def** or **macro**, the result is the union of the free variables of $cddr(x)$. The symbol being *defined* in such a form ($cadr(x)$) can never denote a free variable, exactly *because* it is being defined by the form.

(8) If x is a lambda function, then the variables bound by it will be added to e and with that environment in effect, the result will be the union of the free variables of the body, $cddr(x)$, of the function. (The original environment will be restored after finding the free variables of the function).

Note that all the cases that compute the union of the free variables of some subexpression of x do so by "falling through" to the **while** loop at the end of **freevars()**. The function recurses only through that loop in order to collect free variables of subexpressions.

x	e	Result	Notes
`foo`	`nil`	`(foo)`	
`bar`	`(bar)`	`nil`	*bar* is bound
`(quote foo)`	`nil`	`nil`	
`(f 1 2)`	`nil`	`(f)`	1,2 are not symbols
`(+ x y)`	`nil`	`(x y)`	+ is a primitive operator
`(lambda (x) (f (g x)))`	`nil`	`(f g)`	x is bound in lambda

Fig. 30 – Finding free variables, examples

Figure 30 lists the results of some sample applications of **freevars()**.

```
int     subrp(cell x);

cell freevars(cell x, cell e) {
    cell    n, u, a;
```

```
int     lam;

lam = 0;
if (memq(x, e) != NIL) {
    return NIL;
}
else if (symbolp(x)) {
    return cons(x, NIL);
}
else if (!pairp(x)) {
    return NIL;
}
else if (car(x) == S_quote) {
    return NIL;
}
else if (car(x) == S_apply ||
         car(x) == S_prog ||
         car(x) == S_if ||
         car(x) == S_ifstar ||
         car(x) == S_setq
) {
    x = cdr(x);
}
else if (car(x) == S_def ||
         car(x) == S_macro
) {
    x = cddr(x);
}
else if (car(x) == S_comment) {
    return NIL;
}
else if (subrp(car(x))) {
    x = cdr(x);
}
else if (car(x) == S_lambda) {
    protect(e);
    a = flatargs(cadr(x));
    protect(a);
```

```
        n = set_union(a, e);
        protect(n);
        e = n;
        x = cddr(x);
        lam = 1;
    }
    protect(u = NIL);
    while (pairp(x)) {
        n = freevars(car(x), e);
        protect(n);
        u = set_union(u, n);
        unprot(1);;
        car(Protected) = u;
        x = cdr(x);
    }
    n = unprot(1);
    if (lam) e = unprot(3);
    return n;
}
```

14.2.3 Lambda Functions

This section deals with the transformation of S-expressions representing *functions* ("lambda functions") to S-expressions representing *closures*. It will also introduce new variables by adding their bindings to the top level environment.

The **posq()** function is like **memq()**, but returns the offset of x in a (or **NIL**). It is used to find the location (slot number) of a variable given its symbol.

New symbols are always added to the end of an environment and the offset of a symbol in the environment equals the slot number that will later be used to access the value of the corresponding variable.

Environments are still lists of symbols in this context!

```
int posq(cell x, cell a) {
    int     n;
```

```
    n = 0;
    for (; a != NIL; a = cdr(a)) {
        if (car(a) == x) return n;
        n++;
    }
    return NIL;
}
```

In fact, closure conversion maintains two environments: one containing *bound variables* (*formal arguments*) and another one containing *free variables* in local environments. The top level environment is just the outermost local environment in this context.

Argument symbols are carried along in the variable a and environment symbols in the variable e, where a contains *only* the arguments of the function currently being converted and e contains *only* the free variables of that function.

The **initmap()** function computes an *initmap* for the free variables fv given the arguments a of the immediate outer function (IOF) and the environment e of the IOF. For instance, for $fv = (x\ y)$, $e = (v\ w\ x)$, and $a = (y)$, **initmap()** would return

$((e\ 2\ 0\ x)\ (a\ 0\ 1\ y))$

(The variable names in the initmap will be used in later steps.)

The variables **I_a** and **I_e**, which are defined here (not just declared), will be bound to the symbols **"a"** and **"e"**, which are used to indicate the source vector in initmap entries.

```
cell    I_a, I_e;

cell initmap(cell fv, cell e, cell a) {
    cell    m, n, p;
    int     i, j;

    protect(m = NIL);
    i = 0;
```

```
    while (fv != NIL) {
        p = cons(car(fv), NIL);
        protect(p);
        n = mkfix(i);
        p = cons(n, p);
        car(Protected) = p;
        if ((j = posq(car(fv), a)) != NIL) {
            n = mkfix(j);
            p = cons(n, p);
            unprot(1);
            p = cons(I_a, p);
        }
        else if ((j = posq(car(fv), e)) != NIL) {
            n = mkfix(j);
            p = cons(n, p);
            unprot(1);
            p = cons(I_e, p);
        }
        else {
            error("undefined symbol", car(fv));
        }
        m = cons(p, m);
        car(Protected) = m;
        i++;
        fv = cdr(fv);
    }
    return nreverse(unprot(1));
}
```

The **lastpair()** function returns the last pair (cons cell) of the list x. E.g., the last pair of the list **(x y z)** would be **(z)**. When given **NIL**, the function returns **NIL**.

```
cell lastpair(cell x) {
    if (NIL == x) return NIL;
    while (cdr(x) != NIL)
        x = cdr(x);
    return x;
```

```
}
```

The variable **Env** is bound to the *top level environment*, where "environment" still indicates a list of symbols without bindings. The variable **Envp** always points to the last pair in **Env**. It is the place where new symbols will be added. This is merely a performance hack; maintaining an append pointer is a constant-time operation, while using **conc** to append symbols would result in quadratic time behavior.

```
cell    Env = NIL,
        Envp = NIL;
```

The **newvar()** function adds the symbol x to the top level environment. If the symbol already is in the environment, it will not be added again.

The **newvars()** function adds multiple symbols at once – those contained in the list x.

Top-level bindings are sometimes introduced implicitly by lambda functions, as the following example illustrates:

```
(lambda (x) (foo x))
```

If the variable *foo* has not been defined before, the above expression will do so. However, it does not store any value in the location bound to *foo*, so the symbol *foo* will be bound at this point, but the variable *foo* will still be undefined. Only a slot has been reserved for it.

This is necessary in order to allow for mutually recursive definitions, as in

```
(defun (f x) (if (pair x) (g x)))
(defun (g x) (f (cdr x)))
```

In this program, g is defined implicitly inside of the definition of f. Only *after* defining f, an explicit definition for g is supplied. The value of g will then be stored in the slot that was pre-allocated in the definition of f.

During conversion of lambda functions, **newvars()** will be used to pre-allocate environment slots.

```
void newvar(cell x) {
    cell    n;

    if (memq(x, Env) != NIL) return;
    if (NIL == Envp) Envp = lastpair(Env);
    n = cons(x, NIL);
    cdr(Envp) = n;
    Envp = n;
}

void newvars(cell x) {
    while (x != NIL) {
        newvar(car(x));
        x = cdr(x);
    }
}
```

The **mapconv()** function maps closure conversion over the list x. In terms of LISP, it does

```
(mapcar (lambda (x) (cconv x e a)) x)
```

The **cconv()** function performs the complete closure conversion. It will be defined at the end of this section.

```
cell cconv(cell x, cell e, cell a);

cell mapconv(cell x, cell e, cell a) {
    cell    n, new;

    protect(n = NIL);
    while (pairp(x)) {
        new = cconv(car(x), e, a);
        n = cons(new, n);
        car(Protected) = n;
```

```
        x = cdr(x);
    }
    return nreverse(unprot(1));
}
```

The **lamconv()** ("lambda conversion") function converts an S-expression x, representing a lambda function, to an S-expression representing a closure. The function basically extracts the free variables and arguments of x, builds an initmap, and uses these parts to construct a new S-expression. This process is simple when shown in LISP:

```
(defun (lamconv x e a)
   (let ((fv   (freevars x))
         (args (flatargs (cadr x))))
     (newvars fv)
     @(%closure ,(cadr x)
                ,(initmap fv e a)
                ,@(mapconv (cddr x) fv args))))
```

The corresponding C code has to do a lot of GC protection of intermediate results, though, so it has to be serialized carefully.

The transformation performed by the function is as follows: given $e = (g)$, and $a = (f)$, it converts

```
(lambda (x) (f (g x)))
```

to

```
(%closure (x)
  ((a 0 0 f) (e 0 1 g))
  ((%ref 0) ((%ref 1) (%arg 0))))
```

where the conversion of variable symbols to applications of **%arg** and **%ref** will be done by functions to be discussed later in this chapter.

The **l_closure** variable, which is being defined below, is bound to the symbol **%closure**. This symbol is used to indicate a closure in the S-expression returned by **lamconv()**.

```
cell    I_closure;

cell lamconv(cell x, cell e, cell a) {
    cell    cl, fv, args, m;

    fv = freevars(x, NIL);
    protect(fv);
    newvars(fv);
    args = flatargs(cadr(x));
    protect(args);
    m = initmap(fv, e, a);
    protect(m);
    cl = mapconv(cddr(x), fv, args);
    cl = cons(m, cl);
    cl = cons(cadr(x), cl);
    cl = cons(I_closure, cl);
    unprot(3);
    return cl;
}
```

14.2.4 Lambda Lifting

Lambda lifting [Johnsson1985] is a transformation that converts closures to *combinators*, i.e. to functions without any free variables. The benefit of this transformation is reduced pressure on the *garbage collector*, because no *local environment* has to be created for a function without any free variables. In the ideal case, the application of a combinator will evaluate without allocating any memory at all.

Originally, lambda lifting was called so, because it was supposed to "lift local lambda functions to the global level". However, this explanations seems inaccurate, which is why a different model will be offered here.

A function being "local" or "global" is unrelated to it being a combinator or not. Local functions can be combinators and top level functions can have free variables. This is why the focus in the following explanation will be on the "combinator" aspect of the

function to be transformed and the term “lifting” will be applied to variables instead of functions.

Basically any (pure) function F of the form

```
(lambda (x) (g (h x)))
```

can be transformed to another function F' of the form

```
(lambda (g h x) (g (h x)))
```

given that the proper values for g and h are supplied in every application of F'. These values are exactly the values that F would close over, if the transformation would not take place.

If this transformation is applied to a function F, then *every* place where the function F' is applied to values must be transformed as follows:

```
(F X)   -->   (F' G H X)
```

where X is the value to be bound to x and G and H are the values of g and h that F would close over. This is called a *global program transformation*, because the entire program containing the function F must be scanned for applications of F and all such occurrences must be rewritten to applications of F'. Such a transformation is beyond the scope of the LISP9 compiler.

However, there is a special case that is much easier to transform. When a lambda function is being applied *in situ* (“on the spot”), there is only one single application to modify and the transformation becomes much simpler:

```
     ((lambda (x)      (g (h x))) X)        ; F
-->  ((lambda (g h x) (g (h x))) G H X)     ; F'
```

The bindings of the free variables (g and h in the example) can be extracted from the initmap of F, so supplying the additional arguments G and H is as simple as adding references to the corresponding slots to the function application. The variables g and h are lifted from the environment of the function being transformed to the environment of its *immediate outer function*. Of course, they are still the same variables, but their bindings are contained in a different environment after the transformation. This

is what the **appconv()** function at the end of this subsection will do.

Closure conversion will lambda-lift the variables of all *pure functions* that appear in operator (car) positions of S-expressions. In this context, a “pure” function is a function that does not use **setq** (it may use other effects, such as input/output, though).

The next paragraphs will explain why impure functions cannot be transformed in the way discussed here. Feel free to skip ahead to the next code segment (the **contains()** function), if you are not interested in the details!

Consider the following expression:

```
((lambda (f)
   ((lambda (g)
      (setq f g)
      (f 'a))
    (lambda (x) (f x))))
 nil)
```

It is basically an expanded Scheme **letrec** [R4RS] expression that defines the recursive function f, which will loop forever (given tail call elimination). Its result does not matter, though, the function will only be used to show that it cannot evaluate *at all* when the above expression is being lambda-lifted.

What *should* happen is the following:

Nothing changes in the outermost function, because it does not have any free variables. The other two functions are converted to closures, both closing over the outer argument f. The function **(lambda (g) ...)** then modifies the value of f so it becomes equal to **(lambda (x) ...)**. Because both closures close over the same f, the function (stored in the location) bound to f is now self-referential and hence can recurse:

```
((%closure (f) nil
   ((%closure (g) ((a 0 0 f))
       (setq (%ref 0 f) (%arg 0))
       ((%ref 0 f) 'a))
    (%closure (x) ((a 0 0 f))
```

```
        ((%ref 0 f) (%arg 0)))))
 nil)
```

Of course the binding of f is contained in two different environments after the conversion, but both environments refer to *the same binding* and hence to the same location.

The following is the effect of lambda lifting on the same expression. Again, nothing special happens to the outer function. The innermost function, **(lambda (x) ...)** is also left undisturbed, because it is not being applied in situ.

The drama unfolds in **(lambda (f g) ...)**, formerly **(lambda (g) ...)**, which now receives the outer variable f as an additional argument. Because it does so, it no longer modifies the binding of the outer f, but a local copy instead. **(lambda (x) ...)**, however, still closes over the outer binding, whose value is still **nil** at this point, so f is not a function at all. This contradicts the semantics of the original function.

```
((%closure (f) nil
   ((%closure (f g) nil
      (setq (%arg 0) (%arg 1))
      ((%arg 0) 'a))
    (%arg 0)
    (%closure (x) ((a 0 0 f))
      ((%ref 0 f) (%arg 0)))))
 nil)
```

Because of this specific case, lambda lifting will never be applied to functions containing **setq**.

The **contains()** predicate returns truth, if its argument a, which must be a list, contains its argument x, which must be a symbol or special object.

```
int contains(cell a, cell x) {
    if (a == x) return 1;
    if (pairp(a) &&
        (contains(car(a), x) ||
         contains(cdr(a), x))
    )
```

```
        return 1;
    return 0;
}
```

An expression x is liftable, if it does not contain the symbol **setq**.

Of course, this is a very conservative guess, because expressions may use **setq** in order to modify *bound* variables, and such expressions would still be liftable. The present implementation, though, does not perform any deeper analysis, and so the **liftable()** predicate returns truth, if its argument x does not contain **setq**. (Or even **(quote setq)**!) Feel free to improve this!

```
int liftable(cell x) {
    return !contains(x, S_setq);
}
```

Given an initmap m of a function f, the **liftnames()** function extracts the symbol names of the free variables of f. At a first glance, this may look like a cumbersome way of mimicking **freevars()**, but the same initmap will be passed to **liftargs()** later, where additional information is extracted.

liftnames() extracts the names of additional formal arguments that will be added to a lambda function from which free variables are being lifted. For instance, the function

```
(lambda (x y) (f (g x) (h y)))
```

has the free variables f, g, and h, and so it may have an initmap of the form

((a n **0 f) (e** m **1 g) (e** k **2 h))**

where n, m, and k indicate the source slots of the free variables. When lambda lifting is applied to an application of the above function, the arguments f, g, and h have to be added to its formal argument list. E.g.,

```
((lambda (x y) (f (g x) (h y))) X Y)
```

becomes

```
((lambda (f g h x y) (f (g x) (h y))) F G H X Y)
```

The names of the extra variables are provided by **liftnames()**.

The next step would be to supply the proper arguments in the place of F, G, and H. These arguments are extracted from the initmap m by the **liftargs()** function. Given the above initmap, it would return the references

```
((%arg n) (%ref m) (%ref k))
```

which are the slots from which the environment of f would be populated. Substituting these references for the placeholders F, G, and H in the transformed expression gives

```
((lambda (f g h x y) (f (g x) (h y)))
 (%arg n) (%ref m) (%ref k) X Y)
```

which is exactly the initial application with the variables f, g, and h lifted to the immediate outer function.

The variables **I_arg** and **I_ref**, which are defined below, will be bound to the symbols **%arg** and **%ref**. They represent access to arguments slots and environment slots, respectively.

```
cell liftnames(cell m) {
    #define name cadddr
    cell    a, n;

    protect(a = NIL);
    while (m != NIL) {
        n = name(car(m));
        a = cons(n, a);
        car(Protected) = a;
        m = cdr(m);
    }
    return nreverse(unprot(1));
    #undef name
}

cell    I_arg, I_ref;
```

```
cell liftargs(cell m) {
    #define source  cadr
    cell    a, n;

    protect(a = NIL);
    while (m != NIL) {
        n = source(car(m));
        n = cons(n, NIL);
        n = cons(caar(m) == I_a? I_arg: I_ref,
                 n);
        a = cons(n, a);
        car(Protected) = a;
        m = cdr(m);
    }
    return nreverse(unprot(1));
    #undef source
}
```

The **appconv()** function, finally, performs the complete procedure of lambda lifting. Given the application of an S-expression representing a *lambda function*, it returns an S-expression of the form

((%closure *formals* **nil** *expression* ···**)** *argument* ···**)**

Particularly note that the initmap of the closure is **nil**, which means that it has no free variables and will not allocate a local environment at run time.

The above S-expression is easily created using a **qquote** ("quasiquote") template, as illustrated in the following LISP code: (Note: LISP9 uses "@" in the place of the backquote character.)

```
(defun (appconv x e a)
  (let ((fv (freevars (car x)))
        (fn (car x))
        (as (cdr x)))
    (newvars fv)
    (let ((m (initmap fv e a)))
      @((%closure
```

```
            ,(conc (liftnames m) (cadr fn))
            nil
            ,@(let ((cv (set-union
                              (liftnames m)
                              (flatargs (cadr fn)))))
                (mapcar (lambda (x) (conv x e cv))
                        (cddr fn))))
         ,@(liftargs m)
         ,@(mapcar (lambda (x) (conv x e a))
                   as)))))
```

The arguments of **appconv** are the lambda function application x, the current environment e, and the argument list a, as introduced earlier in this section.

Again, the C code derived from the above LISP function is less comprehensible, because lots of intermediate results have to be GC-protected.

The local variables of the C version of **appconv()** are explained in figure 31.

Variable	Meaning
`fn`	function being applied
`as`	arguments passed to `fn`
`fv`	free variables of `fn`
`fnargs`	formal arguments of `fn` (flattened)
`m`	initmap of `fn` given e and a
`n`	intermediate result
`lv`	variables lifted from `fn`
`vars`	formal argument list of the closure
`cv`	variables bound in the closure (flattened)

Fig. 31 – Local variables of APPCONV

```
cell appconv(cell x, cell e, cell a) {
    cell fn, as, fv, fnargs, m, n, lv, vars, cv;

    fn = car(x);
    as = cdr(x);
```

```
    fv = freevars(fn, NIL);
    protect(fv);
    fnargs = flatargs(cadr(fn));
    protect(fnargs);
    newvars(fv);
    m = initmap(fv, e, a);
    protect(m);
    as = mapconv(as, e, a);
    protect(as);
    n = liftargs(m);
    as = nconc(n, as);
    car(Protected) = as;
    lv = liftnames(m);
    protect(lv);
    vars = conc(lv, cadr(fn));
    protect(vars);
    cv = set_union(lv, fnargs);
    cadr(Protected) = cv;
    fn = mapconv(cddr(fn), e, cv);
    fn = cons(NIL, fn);
    fn = cons(vars, fn);
    fn = cons(I_closure, fn);
    unprot(6);
    return cons(fn, as);
}
```

14.2.5 Global Definitions

The **defconv()** function converts **def** forms as follows:

(def v x**)** $\rightarrow$ **(setq (%ref** n v**)** x**)**

It also creates a new top-level variable that binds the name v to the location n. The binding is established in the deep-binding environment **Glob** at compile time, but the value will be be stored in the location n of the shallow-binding environment at run time.

Note that **%ref** actually has a second argument: the symbol to whose location it refers. This symbol will be used to report undefined variables during evaluation. Such errors occur when a

symbol is bound to a location containing an undefined value, as in

```
; foo is unbound and undefined here
((lambda () foo))
; foo is bound but still undefined here
```

The **def** form will always create a binding with some value stored in its location, though.

```
cell defconv(cell x, cell e, cell a) {
    cell    n, m;

    newvar(cadr(x));
    n = cons(cconv(caddr(x), e, a), NIL);
    protect(n);
    m = mkfix(posq(cadr(x), e));
    protect(m);
    m = cons(I_ref, cons(m, cons(cadr(x), NIL)));
    unprot(2);
    return cons(S_setq, cons(m, n));
}
```

14.2.6 Expression Conversion

The **cconv()** function performs the complete closure conversion of the S-expression x given the local environment e and arguments a. Initially, e will be bound to the top level environment and a will be empty. Both will be updated whenever **cconv()** is called recursively.

cconv() delegates the more complex cases to the above functions, but also handles many simple cases by itself:

The special forms **apply**, **if**, **if***, **prog**, and **setq** will just have their arguments closure-converted. The same applies to primitive functions, like **cons**, **car**, **+**, etc.

Symbols in a will be replaced with **(%arg** n**)**, where n is the argument slot associated with the symbol.

Symbols in e will be replaced with **(%ref** n v**)**, where n is the

environment slot assigned to the symbol and v is the symbol itself.

Symbols neither in a nor in e are undefined and will signal an error.

All non-symbolic atoms are left unchanged.

Applications of the form

(macro v x)

will have their body x converted and otherwise be left unchanged. Macro names are bound in a separate environment that will be invisible at evaluation time.

Comments will be left unchanged.

Function applications will be closure-converted by mapping the conversion over the function itself and all of its arguments.

```
cell cconv(cell x, cell e, cell a) {
    int     n;

    if (pairp(x) &&
        (S_apply == car(x)  ||
         S_if == car(x)     ||
         S_ifstar == car(x) ||
         S_prog == car(x)   ||
         S_setq == car(x)   ||
         subrp(car(x))))
    {
        return cons(car(x),
                    mapconv(cdr(x), e, a));
    }
    if ((n = posq(x, a)) != NIL) {
        return cons(I_arg, cons(mkfix(n), NIL));
    }
    if ((n = posq(x, e)) != NIL) {
        Tmp = mkfix(n);
        n = cons(I_ref, cons(Tmp, cons(x, NIL)));
        Tmp = NIL;
        return n;
```

```
    }
    if (symbolp(x)) {
        error("undefined symbol", x);
        return NIL;
    }
    if (atomp(x)) {
        return x;
    }
    if (S_quote == car(x)) {
        return x;
    }
    if (pairp(car(x)) &&
        S_lambda == caar(x) &&
        liftable(car(x)))
    {
        return appconv(x, e, a);
    }
    if (S_lambda == car(x)) {
        return lamconv(x, e, a);
    }
    if (S_def == car(x)) {
        return defconv(x, e, a);
    }
    if (S_macro == car(x)) {
        return cons(car(x),
                cons(cadr(x),
                 mapconv(cddr(x), e, a)));
    }
    if (S_comment == car(x)) {
        return x;
    }
    return mapconv(x, e, a);
}
```

The **carof()** function returns all car parts (“keys”) of the association list a. I.e., given

$$((key_1 \;.\; value_1) \cdots (key_n \;.\; value_n))$$

it returns

($key_1 \; \cdots \; key_n$)

This function is used to extract the symbol names stored in the global environment **Glob**. The result will be used as the *e* parameter in the above functions.

```
cell carof(cell a) {
    cell    n;

    protect(n = NIL);
    while (a != NIL) {
        n = cons(caar(a), n);
        car(Protected) = n;
        a = cdr(a);
    }
    unprot(1);
    return nreverse(n);
}
```

The **zipenv()** function creates a new (deep-binding) environment from the list of variables *vs* and a previously existing environment *oe*. There may be more variables in *vs* than bindings in *oe*. In this case, **UNDEF** will be stored in the locations of extra slots, thereby indicating that the corresponding variables are still undefined.

For instance, the arguments

vs = **(foo bar baz)**
oe = **((foo . foo-value))**

will result in the new environment

((foo . foo-value) (bar . UNDEF) (baz . UNDEF))

This function is used to convert a list of variables back to a deep-binding environment.

```
cell zipenv(cell vs, cell oe) {
    cell    n, b;
```

```
    protect(n = NIL);
    while (vs != NIL) {
        if (NIL == oe) {
            b = cons(car(vs), cons(UNDEF, NIL));
        }
        else {
            b = car(oe);
            oe = cdr(oe);
        }
        n = cons(b, n);
        car(Protected) = n;
        vs = cdr(vs);
    }
    return nreverse(unprot(1));
}
```

The **clsconv()** function performs the full closure conversion of the S-expression x in the global environment **Glob**. It returns the converted expression and updates the global environment in situ.

When the top level environment is initially empty, the function will add one single undefined slot to it, because symbols will be added destructively to the last pair of **Env**, and this would be impossible, if **Env** was **NIL**.

```
cell clsconv(cell x) {
    cell    n;

    Env = carof(Glob);
    Envp = NIL;
    if (NIL == Env) Env = cons(UNDEF, NIL);
    n = cconv(x, Env, NIL);
    protect(n);
    Glob = zipenv(Env, Glob);
    return unprot(1);
}
```

14.3 The Abstract Machine

The remainder of this chapter will deal with code generation, hence it makes sense to introduce the *target architecture* of the LISP9 compiler at this point. The target architecture of a compiler is the *CPU* and *host environment* for which code is being generated. For native code compilers, the CPU is mostly a machine that is cast in silicon and the host environment is an interface to services supplied by an operating system.

The LISP9 compiler targets a virtual architecture called the *LISP9 abstract machine* (*LAM*), which provides both, instructions for evaluating expressions and services that are typically offered by an operating system. It is called a *virtual* architecture, because it does not exist in hardware, but forms an additional layer of abstraction between the hardware CPU and the program being evaluated.

A LISP program is interpreted by the abstract machine and the abstract machine is interpreted by a CPU. The LAM is a *virtual machine* running on top of a "real" (hardware) CPU. In addition, the LAM maps some of its instructions to operating system services rather than instructions of the CPU. Finally, of course, the operating system is also interpreted by the CPU. See figure 32.

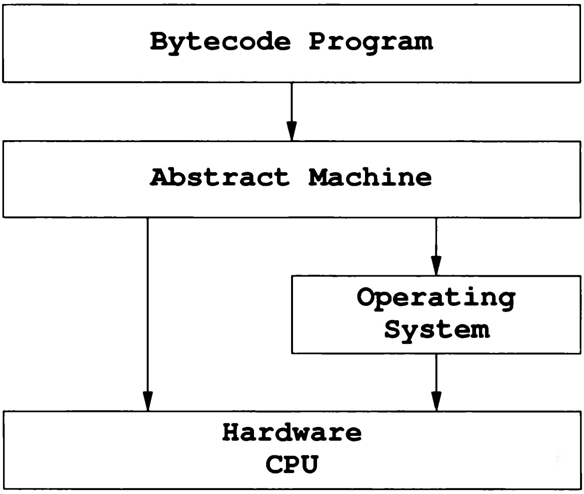


Fig. 32 – Abstraction layers of the LISP9 abstract machine

An abstract machine, like most "real" machines, is defined by its registers, storage, instructions, and other, often language-specific facilities. A common abstract machine for functional programming languages is the *SECD machine* [Landin1964], which consists of four components indicated by the letters of its name:

- stack
- environment
- control
- dump

The "stack" is a push-down list where arguments to instructions and functions as well as intermediate results are stored. The "environment" is an associative array, just like the deep-binding environment of LISP9. "Control" is a list containing the program being evaluated and "dump" is another push-down list where (*stack*, *environment*, *control*) triplets are stored as activation frames for functions whose evaluation has been suspended in order to apply another function. I.e. the "dump" serves as a separate stack holding *continuations*, or points to which control will return after evaluating a function.

The LISP9 abstract machine is only loosely modeled after the SECD machine, which is natural, because the SECD machine is a very abstract model, while the LAM is a concrete implementation. But even then, there are some differences. The LAM has only a single stack that stores both, arguments of function applications and activation frames. The environment is not an associative array, but a set of vectors holding locations, i.e. it uses shallow binding instead of deep binding. Finally, it defines more built-in operations than the original SECD machine.

Many implementations of the SECD machine use data objects that are native to LISP (e.g. [Kogge1991]), maybe because the original design used lambda calculus [Church1941] as its theoretical foundation and/or because it is often used to implement LISP. The LAM also uses LISP data objects in its own implementation. For instance, its stack is a vector, its programs are strings, and its environments are also vectors.

The entire design of the LAM is driven by *simplicity*, *efficiency*, and the attempt to mimic a hardware CPU. Of course, the first two goals are contradictory to some degree, so some compromises have to be made. The last goal is pursued in order to facilitate the translation of LAM code to native machine code.

14.3.1 The Components of the LAM

The LISP9 abstract machine consists of the following components, which are outlined in figure 33:

- an interpreter of LAM instructions (control)
- a set of 7 registers
- a combined control and data stack
- any number of closures
- a top level environment

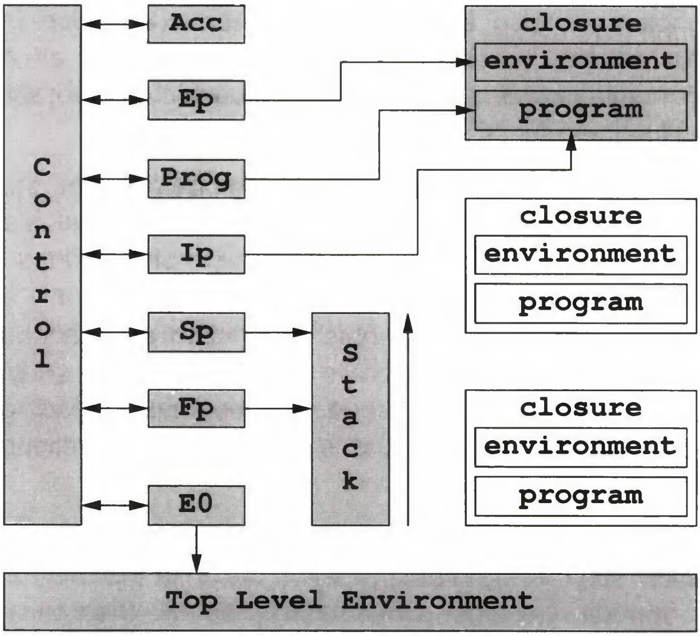


Fig. 33 – The components of the LAM

The definition of the LAM does not include any means of *storage*, except for the stack, the *top level environment* (*TLE*), and the closures. Memory is unbounded in the model: the stack and TLE

can grow indefinitely and there can be any number of closures. In practice memory is managed dynamically, because the data structures used by the machine are subject to garbage collection. In the end, of course, the memory of the LAM is limited either by the underlying hardware or by a static limit imposed during compilation of the LAM.

All data objects are bound to variables, either in the TLE, on the stack, or in one of the closures. The symbol table and object table are implementation details and not important when discussing the LAM, which operates at a more concrete (or less abstract) level than those data structures.

The following *registers* exist in the LISP9 abstract machine:

- **Acc** ("accumulator"), used for arguments and result values
- **Ep** ("environment pointer"), points to the current environment
- **Prog** ("program"), points to the current program fragment
- **Ip** ("instruction pointer"), points to the current instruction in **Prog**
- **Sp** ("stack pointer"), points to the top of the stack
- **Fp** ("frame pointer"), points to the current stack frame
- **E0** ("base environment"), points to the top level environment

All registers of the LAM, except for **Acc**, are used to refer to components of the machine. They are typically used to read and modify those parts of the machine.

The *accumulator* Acc is the only general purpose register of the machine. It is used to exchange values with built-in operators. When an operator has one argument, it receives it in the **Acc** register. When there are multiple arguments, the then *first* one will be passed in **Acc**. In any case, the result of the operation will be returned in the accumulator. When an evaluation finishes, its final result will be stored in **Acc**.

The *environment pointer* **Ep** always points to the environment of the closure whose program is currently evaluating. When no closure is evaluating, it points to the top level environment **E0**.

The *program register* **Prog** always points to the program of the

closure that is currently evaluating. Of course, a LISP program is usually composed of multiple closures. In the context of the LAM, a program (or, more correctly, a *program fragment*) is the code associated with a specific closure. The terms "program" and "program fragment" are used interchangeably in this section.

When no closure is currently evaluating, **Prog** points to a "free" program that is not contained in any closure. This is considered to be an implementation detail, though. As far as the model is concerned, the "free program" could be part of a separate closure that is reserved for this purpose.

The *instruction pointer* **Ip** points to the *next* instruction to interpret inside of **Prog**. Modifying this register passes control to a different part of the same program.

The *stack pointer* **Sp** points to the element most recently pushed to the *runtime stack* (*RTS*). The stack grows "upward", i.e. towards larger values of **Sp**. When the stack is empty, the value of **Sp** is -1. The RTS will grow dynamically during evaluation.

The *frame pointer* **Fp** points *into* the stack frame of the most recently applied function (closure). More specifically, it points to the first argument passed to the function.

A *stack frame* is built whenever a function is being applied to values. The following steps will be performed:

- arguments are pushed to the stack in *reverse* (right-to-left) order
- the number of arguments is pushed to the stack
- the environment pointer is pushed to the stack
- the return continuation is pushed to the stack
- the frame pointer is pushed to the stack
- the frame pointer is pointed to the first argument

These steps leave the top of the run time stack in the configuration depicted in figure 34.

Each *closure* in the LAM consists of two components called its *program fragment* and its *local environment*. The local environment is an environment that has been constructed during

closure conversion. (Or, rather, *instructions* for building that environment have been generated during conversion.)

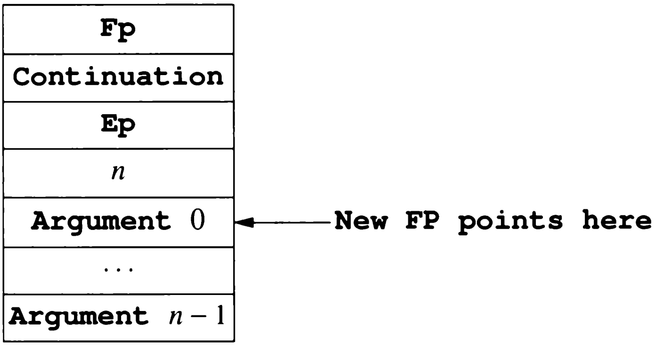


Fig. 34 – Stack frame layout

Whenever a closure is being invoked by passing control to its *program fragment*, the local environment of the closure becomes the current environment, and when the closure is done evaluating, the environment what was in effect before will be restored.

The program fragment of a closure contains instructions for performing a computation. These instructions will be discussed in the next subsection.

In the present implementation, closures contain another piece of information, and that is their *entry point*. Due to the way in which the LISP9 compiler emits code, program fragments will contain instructions that are not part of the computation performed by a closure. The entry point marks the first instruction that actually belongs to the code of the closure.

14.3.2 The Instruction Set of the LAM

The instructions of the LISP9 abstract machine will be described semi-formally in this subsection, using an approach that is loosely based on structural operational semantics [Plotkin1981/2004].

Structural operational semantics describes the meaning of a model of program evaluation at the "microscopic" level, by specifying exactly what each instruction does and how it ts being interpreted. In order to accomplish this goal, the following notation shall be introduced.

S	the runtime stack
OBTAB	the object table
x	lower case names are auxiliary variables
$(A, B, \cdots)$	the tuple containing $A, B, \cdots$
$(A..B)$	the tuple enumerating $A, \ldots, B$
A_B	the Bth element of the list or tuple A
$[A]$	a fresh box containing A
$\|A\|$	the length of the list A
$C(i, e, p)$	a closure with entry point i, environment e, and program p
$C_E(A)$	the environment of the closure A
$C_P(A)$	the program fragment of the closure A
$C_0(A)$	the entry point of the closure A
$A \leftarrow B$	set A to the value of B
$A[B]$	the Bth element of the vector A; environments are vectors of boxes
$A[[B]]$	the value stored in the Bth box of the vector A; bindings are boxes
$A; B$	first do A, then do B
$A\downarrow$	short for $Sp \leftarrow Sp + 1;\ \mathbf{S}[Sp] \leftarrow A$
$A\uparrow$	short for $A \leftarrow \mathbf{S}[Sp];\ Sp \leftarrow Sp - 1$
$A[B] \leftarrow C[D{:}\ E]$	copy E vector elements from $C[D]$ through $C[D + E - 1]$ to vector elements $A[B]$ through $A[B + E - 1]$
if A; B else C *end*	if A is true, do B, else do C; "else C" is optional
for each A in B; C end	do C with A bound to each member of list B, from left to right

For instance,

- $Acc \leftarrow 0$ would store 0 in the accumulator.
- $\mathbf{S}[Sp] \leftarrow nil$ would deposit nil in the top stack element. It would *not* push **nil**, but simply change $\mathbf{S}[Sp]$ to **nil**.
- $Ep[0]$ would indicate the first binding in the current environment.
- $Ep[[0]]$ would indicate the value stored in $Ep[0]$.
- $Fp\downarrow$ would push the value of the Fp register to the stack.

- $Acc\uparrow$ would pop a value off the stack and place it in the accumulator.
- $A[0] \leftarrow B[1\!:\!3]$ would copy $B[1]$ to $A[0]$, $B[2]$ to $A[1]$, and $B[3]$ to $A[2]$. Assignment is parallel, so regions may overlap.

Furthermore, functions like **cons**, **conc** or **list** are sometimes used in formal definitions of instructions. E.g.,

- $Acc \leftarrow cons(\mathbf{nil}, \mathbf{nil})$ would store **(nil)** in the accumulator.

The following definitions are presented at a very abstract level, and some details are being glossed over. In particular, integers are being pushed the stack, which is not possible in the given infrastructure without packaging them in a fixnum object.

Using the above notation, the LAM instructions are defined as follows:

Load Instructions

These instructions load values from various sources into the accumulator. The notation *quote n m* indicates that the *quote* instruction occupies three bytes: the instruction itself and two argument bytes, which form one value, $256n + m$.

$quote\ n\ m$:	$Acc \leftarrow \mathbf{OBTAB}[256n + m]$

Load a literal object from the object table.

$arg\ n\ m$:	$Acc \leftarrow \mathbf{S}[[Fp - (256n + m)]]$

Load the $(256n + m)^{th}$ argument from the current stack frame.

$ref\ n\ m\ j\ k$:	$Acc \leftarrow Ep[[Fp - (256n + m)]]$

Load the value from the $(256n + m)^{th}$ binding of the current environment. The value $256j + k$ points to the symbol table and is not used by the LAM.

Store Instructions

$setarg\ n\ m$:	$\mathbf{S}[[Fp - (256n + m)]] \leftarrow Acc$

Store Acc in the $(256n + m)^{th}$ argument binding of the current stack frame.

setref n m: $Ep[[256n+m]] \leftarrow Acc$

Store Acc in the $(256n+m)^{th}$ binding of the current environment.

Stack Instructions

push: $[Acc]\downarrow$

Push a fresh *box* (*location*) containing Acc.

pushval n m: $(256n+m)\downarrow$

Push the value $256n+m$ without a surrounding box.

pushtrue: $T\downarrow$

Push the canonical "true" value.

pop: $Acc\uparrow$

Pop a value off the stack and place it in Acc.

drop: $Sp \leftarrow Sp - 1$

Pop a value off the stack and forget it.

Jump and Branch Instructions

jmp n m: $Ip \leftarrow (256n+m)$

Pass control to the $(256n+m)^{th}$ byte of $Prog$.

brf n m: if $Acc = \textbf{nil}$; $Ip \leftarrow (256n+m)$ end

Jump to $256n+m$ if Acc is **nil** ("branch on false").

brt n m: if $Acc \neq \textbf{nil}$; $Ip \leftarrow (256n+m)$ end

Jump to $256n+m$ if Acc is not **nil** ("branch on true").

halt: halt evaluation, result in Acc

Some instructions are described informally, because it does not make sense to invent some extra notation for them.

Function Application Instructions

The following instructions will be involved in *function application*. The instructions that apply a function to arguments come in two

flavors, a ***recursive*** one and a ***tail-recursive*** one. The recursive variants (*apply*, *applis*) will create a new stack frame, while the tail-recursive ones (*tailapp*, *applist*) will reuse an existing stack frame, thereby making function application a constant-space operation.

There is quite a bit of complexity hidden in tail-recursive function application instructions, which is why the recursive variants will be introduced first.

apply:	$Ep\downarrow;\ (Ip+1,\ Prog)\downarrow;\ Ep \leftarrow C_E(Acc);$ $Prog \leftarrow C_P(Acc);\ Ip \leftarrow C_0(Acc)$

Apply the closure in *Acc* to the arguments on the stack. The number of arguments is expected on the top of the stack at this point. The *apply* instruction itself will push *Ep* and the return continuation $(Ip + 1, Prog)$. The frame pointer will later be pushed by *enter* or *entcol*, thereby completing the stack frame shown in figure 34. The instruction sets up a new current environment, program fragment, and instruction pointer from the closure in *Acc*, thereby transferring control to the closure.

enter n m:	if $\mathbf{S}[Sp-2] \neq 256n+m$; **error** else $Fp\downarrow;\ Fp \leftarrow Sp-4$ end

In the partial stack frame present when *enter* is interpreted, the number of arguments on the stack is located in $\mathbf{S}[Sp-2]$. When this value is not equal to $256n+m$, an error is signalled. When the number of arguments matches the operand of *enter*, *Fp* will be saved on the stack and then pointed at the first argument in the new stack frame.

return:	$Fp\uparrow;\ c\uparrow;\ Ip \leftarrow c_{0;}\ Prog \leftarrow c_1;\ Ep\uparrow;$ $n\uparrow;\ Sp \leftarrow Sp-n$

Reverse the effect of *apply* and *enter* (or *entcol*): Restore *Fp*, *Ip*, *Prog*, *Ep*, and remove the arguments and argument count from the stack. This instruction will remove an entire stack frame and pass control to the instruction following the *apply* instruction that started building the stack frame.

tailapp: $Ep \leftarrow C_E(Acc);\ \ Prog \leftarrow C_P(Acc);$
$m \leftarrow \mathbf{S}[Sp];\ \ n \leftarrow \mathbf{S}[Sp - m - 4];$
$i \leftarrow Sp - m;\ \ j \leftarrow Sp - m - n - 4;$
if $n = m;$
  $\mathbf{S}[j] \leftarrow \mathbf{S}[i{:}\, m + 1];$
  $Sp \leftarrow Sp - n - 1;\ \ Fp\uparrow$
else
  $e \leftarrow \mathbf{S}[Sp - m - 3];\ \ k \leftarrow \mathbf{S}[Sp - m - 2];$
  $Fp \leftarrow \mathbf{S}[Sp - m - 1];$
  $\mathbf{S}[j] \leftarrow \mathbf{S}[i{:}\, m + 1];$
  $Sp \leftarrow Sp - n - 2;$
  $\mathbf{S}[Sp - 1] \leftarrow e;\ \ \mathbf{S}[Sp] \leftarrow k$
end;
$Ip \leftarrow C_0(Acc)$

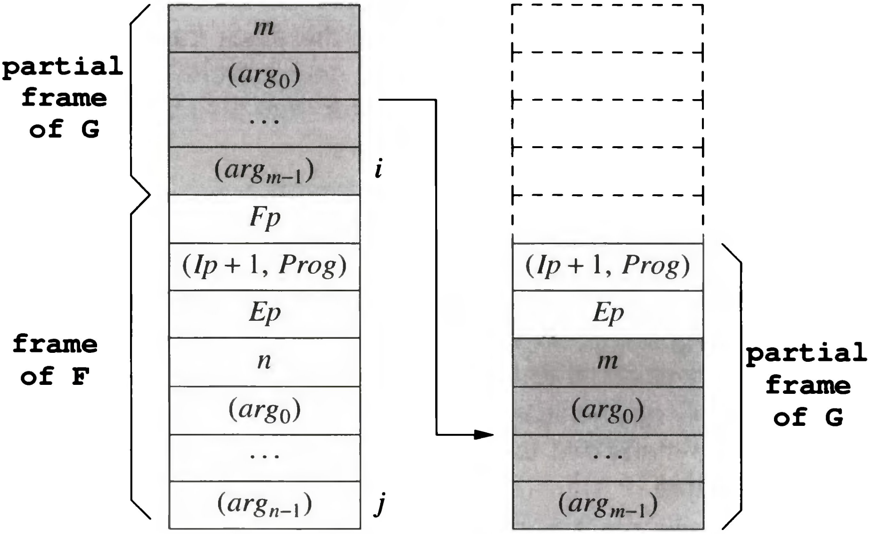


Fig. 35 – Tail application, simple case ($n = m$)

For tail-recursive application – or just *tail application* – there are two cases, a simple one and a not-so-simple one. The simple one occurs when the applying function F has the same number of arguments as the function G that is *being* applied ($n = m$). In this case, the argument bindings of G are copied to the argument slots of F, the arguments of G are removed, and Fp is restored from F's stack frame (because it will later be saved by *enter*). This case is depicted in figure 35.

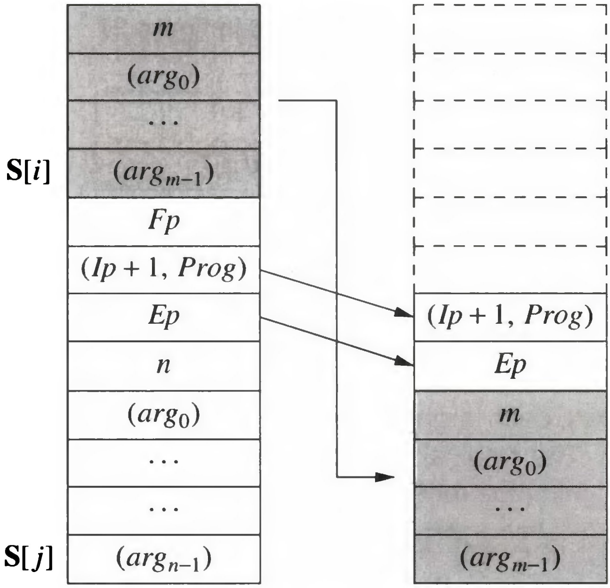


Fig. 36 – Tail application, complex case ($n > m$)

The more complex case occurs when the applying function F has more or fewer arguments than the applied function G. In this case, the stack frames are of different size and hence the return continuation and the saved environment of F have to be moved out of the way before copying the arguments and moved back into place when done. This is what happens in the "else" branch of *tailapp*. m is the argument count of G and n is the argument count of F. S[i] is the first stack element that belongs to the (partial) frame of G and S[j] is the first element of the frame of F. The $n > m$ case is illustrated in figure 36. If $n < m$, the continuation and environment pointer would have to move in the opposite direction (upward).

Note that in all cases described here, by copying the arguments of G to the frame of F, the frame of F becomes the frame of G.

Building Argument Lists

The (recursive) *applis* and (tail-recursive) *applist* instructions both make use of an auxiliary definition named *arglis*, which builds an argument frame on the stack from a LISP list. Like *apply* and *tailapp*, these instruction expect a closure in *Acc*. Unlike those,

they expect the arguments to be passed to *Acc* in a single list on top of the stack. The effect of *arglis* is depicted in figure 37.

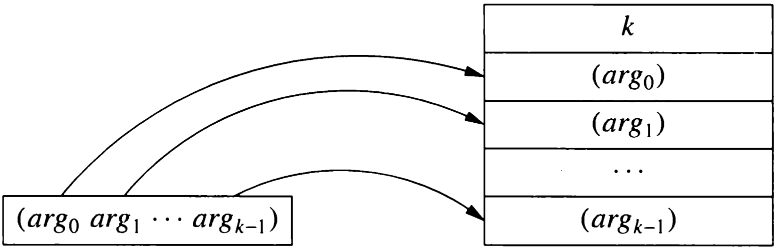


Fig. 37 – Building an argument list

Note that *arglis* *boxes* each argument while transferring it to its individual stack slot. The original list argument of *arglis* will be overwritten by (arg_{k-1}) in the process.

arglis: $a \leftarrow \mathbf{S}[Sp]$; $k \leftarrow |a|$; $Sp \leftarrow Sp + k; i \leftarrow Sp - 1$;
for each x in a;
$\mathbf{S}[i] \leftarrow [x]$; $i \leftarrow i - 1$
end;
$\mathbf{S}[Sp] \leftarrow k$

applis: *arglis*; *apply*

applist: *arglis*; *tailapp*

Variable Argument Lists

Sometimes the number of arguments passed to a function cannot be known at compile time, because the function accepts some *optional arguments*, like z in

```
(lambda (x y . z) z)
```

In these cases, the *optional arguments* passed to the function have to be picked from the stack and packaged in a list. This is what the *entcol* ("enter collecting") instruction does.

Figure 38 illustrates the process. The k *fixed arguments* arg_0 through arg_{k-1} are left as they are, but any optional arguments arg_k through arg_{n-1} are collected in a list. The total number of actual arguments is n. The list containing all optional arguments finally replaces the first optional argument.

```
entcol n m:   k ← 256n + m;  q ← S[Sp − 2];
              if q < k;  error  end;
              a ← nil;  i ← Sp − k − 3;
              for each x in (0 .. q − k)
                 a ← conc(a, list(S[[i]]));
                 i ← i − 1;
              end;
              if q > k;
                 S[Sp − k − 3] ← [a]
              else
                 nil↓;  s ← Sp − q − 3;  d ← Sp − q − 2;
                 S[d] ← S[s: q + 3]
                 S[Sp − 2] ← k + 1;  S[Sp − k − 3] ← [nil];
              end;
              Fp↓;  Fp ← Sp − 4
```

Of course this works only as long as any optional arguments are being passed to the function, because otherwise there is no argument slot on the stack in which the empty list of optional arguments could be stored. This special case creates a lot of additional complexity in the *entcol* instruction.

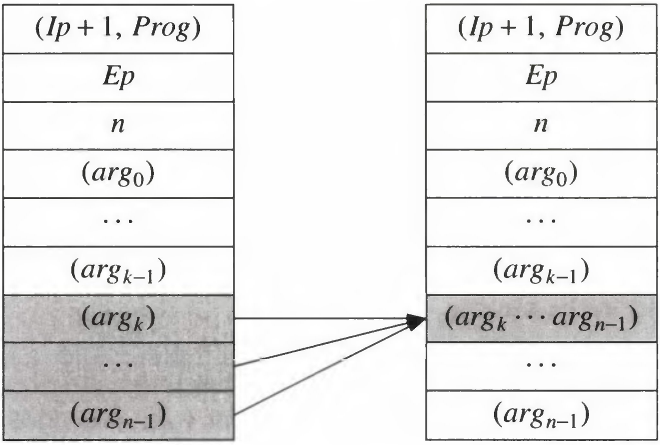


Fig. 38 – Collecting optional arguments

In order to create space for the empty list, the entire (partial) stack frame has to move "up" (toward higher addresses) by one slot so that the additional argument can be inserted at its bottom. In

addition, the argument count in the frame has to be adjusted, so that a *ret* instruction can remove the additional argument properly when the function returns. See figure 39.

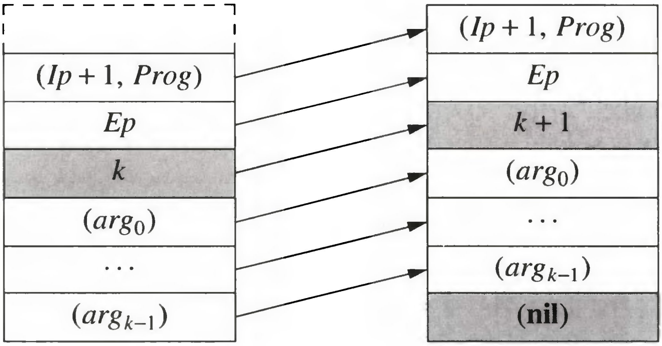


Fig. 39 – Collecting optional arguments, special case

The *entcol* instruction also makes sure that there is an actual argument for each fixed argument. The number of fixed arguments is $k = 256n + m$ and the number of actual arguments is taken from the stack: $q = \mathbf{S}[Sp-2]$.

The "for each" loop collects the optional arguments from the stack into the list a. Then, if $q > k$, the binding $[a]$ is simply placed in an now-unused slot of the frame.

If $q = k$, the partial stack frame is moved up, where s is the source and d the destination address for moving the block of slots. Finally, the argument count ($\mathbf{S}[Sp-2]$) is adjusted and the additional [**nil**] binding is inserted.

Like *enter*, *entcol* saves the frame pointer of the caller and adjusts Fp to point into the new frame.

Closure Building Instructions

closure n m:	$Acc \leftarrow C(256n+m, Acc, Prog)$

Create a fresh *closure* with entry point $256n+m$, the local environment taken from Acc, and the program fragment taken from the current program, $Prog$.

Each closure is created by a code fragment of the form

```
mkenv          k
cparg/cpref    ... 0
cparg/cpref    ... k-1
closure        i
```

where k is the number of free variables in the corresponding function and i is the entry point of the bytecode of the closure. A series of *cparg* and *cpref* instructions is used to populate the environment created by *mkenv*.

When the closure is a combinator, *propenv* is used in the place of *mkenv* and there are no *cparg* or *cpref* instructions.

$mkenv\ n\ m$: $\quad Acc \leftarrow mkvec(256n + m)$

Create a fresh environment of the size $256n + m$. Because an environment is implemented as a vector, this is the same as an application of **mkvec()**. The *mkenv* instruction creates an environment for the *closure* instruction.

$propenv\ n\ m$: $\quad Acc \leftarrow Ep$

Copy the current environment to Acc so that a subsequent *closure* instruction will use it when building a *combinator*. This is intensionally the same as *mkenv* 0, but more efficient, because it does not do a function call.

$cparg\ n\ m\ j\ k$: $\quad Acc[256j + k] \leftarrow \mathbf{S}[Fp - (256n + m)]$

Copy a *binding* (not a value!) from argument slot $256n + m$ to slot $256j + k$ of the fresh environment pointed to by Acc.

$cpref\ n\ m\ j\ k$: $\quad Acc[256j + k] \leftarrow Ep[256n + m]$

Copy a binding (not a value!) from environment slot $256n + m$ to slot $256j + k$ of the fresh environment pointed to by Acc.

Non-Local Transfer Instructions

$catch$: $\quad c \leftarrow (Ip + 2, Sp, Fp, Ep, Prog);\ \ c\downarrow$

Capture the current *context*, store it in a *catch tag*, and push that tag to the stack. The current context is the point at which program execution will continue after finishing interpretation of the *catch* instruction itself.

Note that the information captured here is not sufficient to capture the "current continuation" in the sense of Scheme's **call-with-current-continuation** function [Landin1965] [R4RS]. The *catch tag* returned by the *catch* instruction only reifies an *escape continuation*, i.e. a continuation that can only be activated *once* and only from within the *dynamic extent* of the corresponding application of the **catch** function.

throw:	$c \leftarrow Acc;\ Acc\uparrow;\ Prog \leftarrow c_4;\ Ep \leftarrow c_3;$ $Fp \leftarrow c_2;\ Sp \leftarrow c_1;\ Ip \leftarrow c_0$

Reinstate the context pointed to by Acc that was captured previously by *catch*. The value in $\mathbf{S}[Sp]$ will be transferred to Acc, thereby making the *catch* instruction return that value after interpreting *throw*. The local variable c is used to hold the *catch tag* while extracting the register values from it.

The *throw* instruction transfers control *non-local transfer non-locally*, that is, not only to a different location in the $Prog$ vector (or even a different program fragment), but also to a different context or *stack frame*. The frame to which control is being transferred must still be on the stack when *throw* is being interpreted.

Macro Binding Instruction

The *macro binding* instruction (*macro*) is not really part of the LAM, because it binds the value in Acc to a special location that is not accessible otherwise during evaluation. From the LAM's perspective, that location would be *write-only*.

The "special location" described here is an environment in which *macro bodies* (functions) are bound to symbols. This environment is only visible at macro expansion time, which occurs after reading an expression but before compiling it.

macro n m:	bind macro body Acc to symbol in SYMTAB slot $256n + m$

The argument $256n + m$ is an offset into **Symbols**, which is bound to a vector containing all symbols known to the system. Acc must be bound to a closure at this point or an error will be signalled.

Interpreter Loop

The registers of the abstract machine will be set up as follows when interpretation starts:

$$
\begin{aligned}
Prog &= \text{program to interpret}\\
Ip &= 0\\
Acc &= \textbf{nil}\\
E0 &= (listvec\ (map\ cdr\ \textbf{Glob}))\\
Ep &= E0\\
Sp &= -1\\
Fp &= -1
\end{aligned}
$$

This initial configuration can be thought of as the starting point of the interpretation of the body of a closure with $C_0 = 0$, $C_E = E0$, and C_P equal to the bytecode to interpret. Interpretation then commences as follows:

- An instruction is fetched from the Ip^{th} byte of $Prog$ and Ip is incremented by 1.
- If the instruction has at least one argument, it is fetched from the two bytes following the instruction and Ip is incremented by 2.
- If the instruction has two arguments, the second argument is fetched from the two bytes following the first argument and Ip is once again incremented by 2.
- The *microcode* of the instruction is interpreted.
- The entire process starts over.

The microcode of an instruction is a C program fragment that implements that instruction. For the instructions discussed here so far, the C code implements the operational semantics presented in this subsection.

Inline Functions

Finally, there is a set of LAM instructions that are used for inlining LISP functions. For instance, the *cons* instruction implements the **cons** function. All instructions that implement LISP functions expect their arguments on the stack and in the accumulator, where the accumulator holds the *first* argument. When there is only a single argument, it will be passed in the accumulator.

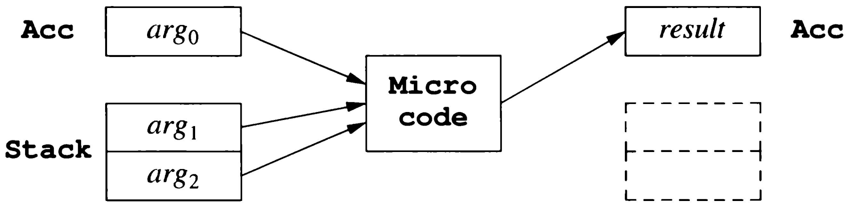


Fig. 40 – Inline function application with 3 arguments

For instance, the *not* instruction would expect its single argument in *Acc*, the *cons* instruction would expect its first argument in *Acc* and its second one in $\mathbf{S}[Sp]$, and the *substr* instruction would expect its first argument *Acc*, its second one in $\mathbf{S}[Sp]$, and its third one in $\mathbf{S}[Sp-1]$. All of these instructions remove their arguments from the stack (if any) and return their result in *Acc*. See figure 40.

There are no *variadic* instructions – functions with variable numbers of arguments will be handled by the compiler and translated to instructions with a fixed number of arguments.

14.4 Code Generation

14.4.1 Object Table Management

Bytecode is a very compact representation for abstract machine code, and it is also used in the LISP9 implementation. The code, as its name suggests, consists of a sequence of bytes. One early choice to make when designing the layout of the code is how to represent *literal data objects*. For instance, how can the following objects be expressed *as a sequence of bytes*?

```
#\x  123  foo  "some string"  ((a b) (c d) (e f))
```

Characters, at least when they are single-byte entities, are trivial; they can just be inlined in the stream of instructions. Integers are a bit more challenging. Of course, they could be stored in some fixed byte order, but what about other numeric data types, like real numbers, rationals, or *bignum* (unbounded) integers?

One could possibly get away with inlining strings in bytecode, but what about lists? Or vectors? Serializing them and reconstructing them at run time would add an unacceptable cost to the process of evaluation.

So there are basically two choices:

(1) Use a abstract machine code representation that allows to inline literal data objects. Of course this is not possible in real *byte*-code; larger objects have to be used instead of bytes.

(2) Use a separate storage area to store data objects and make bytecode refer to that area.

Let us consider option (1) for a moment. The LISP program

```
(+ 1 2 3)
```

would compile to code that might look like this:

```
quote 3
push
quote 2
+
push
quote 1
+
```

The meaning of the program do not matter at the moment, but there are 10 slots in it: 7 instructions and 3 arguments. In order to be able to inline data objects, the program has to be stored in a LISP vector, so all objects stored in that vector have to be LISP objects. Even instructions, which are typically small integers, have to be stored as fixnums. Fixnums have a size of 2 nodes each. So there are two nodes and one vector slot per instruction plus one vector slot per literal data object.

Basically, the code fragment containing the above program could be thought of as:

```
#(quote 3 push quote 2 + push quote 1 +)
```

See figure 41 for an illustration.

In LISP9 on a 32-bit machine, one node allocates 9 bytes and a vector cell 4 bytes, so each instruction uses $2 \times 9 + 4 = 22$ bytes. Seven instructions plus three literals give $7 \times 22 + 3 \times 4 = 166$ bytes, excluding the space for the literals **1**, **2**, and **3** themselves.

This is a lot of space for such a small program, but this approach would completely eliminate the need to maintain an *object table*,

as it was discussed in the memory management chapter. If space is not a concern, this approach is absolutely feasible.

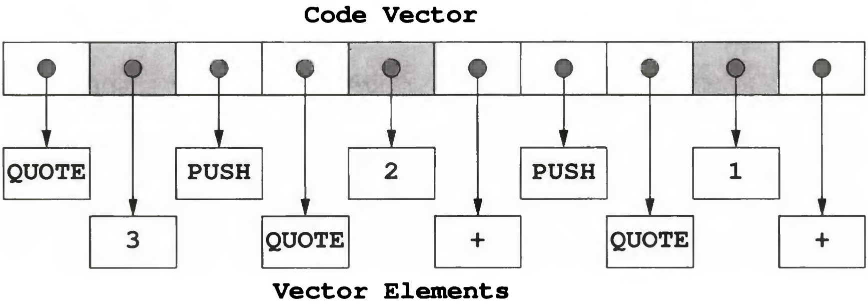


Fig. 41 – Abstract machine code with inlined data objects

Approach (2) would store literal data objects in a separate vector called the object table. When data objects are stored in a separate vector, bytecode can refer to those objects by their offset in the table. Hence bytecode can be mapped to a vector of bytes. In LISP9, strings are vectors of bytes, so programs will be stored in strings. See figure 42. References to data objects are encoded in two subsequent bytes, allowing for $65,536$ data objects in total. If more space is required, argument sizes could be extended to 3 or 4 bytes.

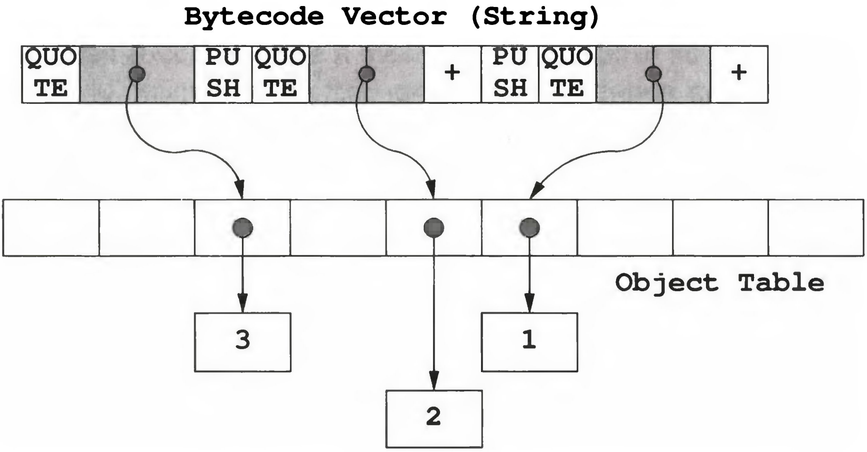


Fig. 42 – Bytecode with data objects in an object table

When using this method, instructions have a size of a single byte and references a size of two bytes, giving $7 \times 1 + 3 \times 2 = 13$ bytes

for the same program, which is more than an order of magnitude smaller than the code size of method (1). The advantage is diminshed a little bit, though, because method (2) requires to maintain an object table, which has a size of at least $5 \times CHUNKSIZE$ bytes and grows by multiples of **CHUNKSIZE** when necessary.

The variable **Obarray** will be bound to a vector that implements the object table and **Obmap** is a bytevector that holds the current state of each slot of the object table. These states are "free", "allocated", and "used"; they have been explained in detail in the chapter about memory management.

The variable **Obptr** points to the next free slot in **Obarray**. The pointer starts at the first slot and then keeps cycling through the table. The idea behind this approach is that slots are often freed in chunks, so remembering the last free slot will probably lead to another free slot in constant time in the next allocation. In the worst case – few free slots scattered across the table – lookup time will be linear.

The variable **Obhash** will be bound to a hash table that maps objects to object table slots. It is used for deduplication of frequently used objects. Without this hash table, the object table would get spammed with lots of instances of frequently used objects, like **nil**, **0**, **1**, **x**, **y**, **#\sp**, etc.

The present implementation is probably deduplicating a bit too aggressively. It merges equal instances of fixnums, characters, and strings, thereby making them equal in the sense of **eq**. This is a controversial idea, because it can encourage the use of **eq**, although the definition of the function **eq** [CLtL1984] and **eq?** [R4RS] emphatically states that the result of such an operation is *undefined* in the above cases.

In LISP9, this implementation choice causes results that may not be expected by some programmers, such as

```
(eq 5 5)          =>  t
(eq 5 (+ 2 3))    =>  nil
```

and

```
(eq "foo" "foo")                =>  t
(eq "foo" (scopy "foo"))        =>  nil
```

(Note that all of the four expressions above have undefined results in LISP9!) Without deduplication, though, the object table might grow at a much faster rate. The reader is advised to conduct their own experiments in this matter.

```
cell    Obhash = NIL,
        Obarray = NIL,
        Obmap = NIL;
int     Obptr = 0;
```

The **obslot()** function searches a free slot in the object table and allocates it. In the innermost loop, it scans the table by moving **Obptr** one time through the entire table, looking for an unused slot. When no such slot is present, a *garbage collection* is performed and then the scan is repeated.

When still no free slot is found, the object table size is increased by **CHUNKSIZE** slots and then the entire process is repeated from the beginning. This is done in the outer loop. When the table size grows beyond $2^{16} = 65,536$ slots, an error is signalled. This is necessary, because references to object table slots in bytecode have a size of 16 bits.

In case the bytecode argument size should ever be changed, this aspect of the **obslot()** function will have to adjusted accordingly.

obslot() returns the index of the allocated slot.

```
int obslot(void) {
    int     i, j, k, m;
    byte    *s;
    cell    n;

    for (m = 0; m < 2; m++) {
        for (j = 0; j < 2; j++) {
            k = veclen(Obarray);
            s = string(Obmap);
```

```
            for (i=0; i<k; i++) {
                if (OBFREE == s[Obptr]) {
                    s[Obptr] = OBALLOC;
                    return Obptr;
                }
                Obptr++;
                if (Obptr >= k) Obptr = 0;
            }
            if (0 == j) gc();
        }
        if (k + CHUNKSIZE >= 64 * 1024) break;
        n = mkvec(k + CHUNKSIZE);
        memcpy(vector(n), vector(Obarray),
                k * sizeof(cell));
        Obarray = n;
        n = mkstr(NULL, k + CHUNKSIZE);
        memset(string(n), OBFREE, k+CHUNKSIZE);
        memcpy(string(n), string(Obmap), k);
        Obmap = n;
    }
    error("out of object space", UNDEF);
    return -1;
}
```

The **obindex()** function returns an object table (*OBTAB*) index for the S-expression x. For *pairs*, *vectors*, and *closures*, it returns a fresh slot immediately. All other objects will be deduplicated using the **Obhash** table.

When the object x exists as a key in the **Obhash** table *and* the corresponding OBTAB slot is in use *and* x matches the object in the OBTAB slot, then the slot number is returned, thereby reusing an existing object in the object table.

If the object x exists as a key in **Obhash** and the corresponding slot is unused *or* contains a different object, then the object x is removed from the **Obhash** table. Such stale entries can exist in **Obhash**, because the garbage collector cleans up the OBTAB, but not the hash table.

If the object x either has been removed or was not present in the hash table from the beginning, a fresh slot will be allocated and associated with the object by adding it to the **Obhash** table.

Of course this approach it not optimal, because stale object may stay in the hash table longer than necessary. It does, however, improve the performance of the garbage collector and abstract machine, because it limits **Obhash** cleanup to compile time.

```
int obindex(cell x) {
    cell    n;
    int     i;

    if (pairp(x) || vectorp(x) || closurep(x))
        return obslot();
    n = htlookup(Obhash, x);
    if (n != UNDEF) {
        i = fixval(cdr(n));
        if (string(Obmap)[i] != OBFREE &&
            match(x, vector(Obarray)[i])
        )
            return i;
        htrem(Obhash, x);
    }
    i = obslot();
    htadd(Obhash, x, mkfix(i));
    return i;
}
```

14.4.2 Bytecode Generation

The LISP9 system stores the bytecode of its abstract machine in *bytecode containers*, which are strings with an additional **T_BYTECODE** type tag attached, as shown in figure 43.

The additional type tag is a hint for the *garbage collector* that tells it to scan the attached string for references to the object table. See the chapter on memory management for details.

The variable **Emitbuf** ("emit buffer") will be bound to a bytecode

container to which the compiler will emit its code. The variable **Here** is the offset of the first bytecode address that does not (yet) contain any code. The emit buffer will grow in size (by **CHUNKSIZE** bytes) when it runs full.

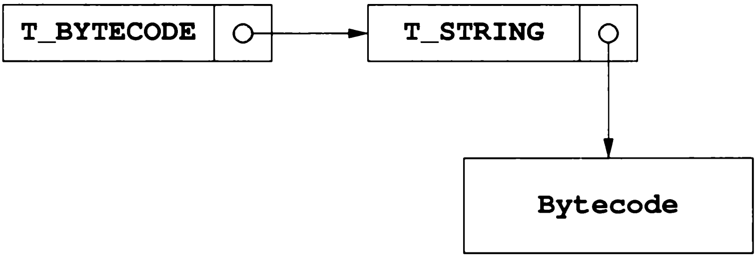


Fig. 43 – Bytecode container

Because the part of the compiler that is discussed in this section *emits* bytecode instructions, it is also called the *emitter*. It is at the opposite end of the compilation process than the parser. (See figure 26, page 127.)

```
cell    Emitbuf = NIL;
int     Here = 0;
```

The **emit()** function emits a single byte of bytecode and advances the **Here** pointer. When the emit buffer runs full, it first allocates a new buffer and copies the code emitted to far to that new buffer and then replaces the internal string of **Emitbuf** in situ.

```
void emit(int x) {
    cell    n;
    byte    *vp, *vn;
    int     i, k;

    if (Here >= stringlen(cdr(Emitbuf))) {
        protect(x);
        k = stringlen(cdr(Emitbuf));
        n = mkstr(NULL, CHUNKSIZE + k);
        vp = string(cdr(Emitbuf));
        vn = string(n);
```

```
        for (i = 0; i < k; i++) vn[i] = vp[i];
        cdr(Emitbuf) = n;
        unprot(1);
    }
    string(cdr(Emitbuf))[Here] = x;
    Here++;
}
```

The **emitop()** macro emits a bytecode instruction, which is always a single byte.

The **emitarg()** function emits a 16-bit *inline operand* in big-endian (high byte, low byte) order. Inline operands are encoded in the instuction stream. They can be, for example, references to object table slots in *quote* instructions, bytecode addresses in jump and branch instruction, references to environment slots in *ref* instructions, etc. All inline operands are unsigned. (This means that **sizeof(int)** has to be greater than 2.)

```
#define emitop(op) emit(op)

void emitarg(int i) {
    if (i < 0 || i > 65535)
        error("bytecode argument out of range",
              mkfix(i));
    emit(i >> 8);
    emit(i & 255);
}
```

The **emitq()** function emits a reference to the *literal data object* x in the object table. It obtains the index value for the object from the **obindex()** function (q.v.).

Note that it is also the **emitq()** function that *adds* the object to the object table. That is, the object table is completely built at compile time. At run time, it just serves as a shallow-binding data structure containing objects. Hence loading a literal data object is a constant-time operation.

```
void emitq(cell x) {
    int     i;

    i = obindex(x);
    vector(Obarray)[i] = x;
    emitop(OP_QUOTE);
    emitarg(i);
}
```

The **patch()** function changes the 16-bit quantity in locations a and $a+1$ of the emit buffer to the value n. It is used for back-patching jump and branch instructions. See the **compif()** and **compcls()** functions later in this section for details.

```
void patch(int a, int n) {
    if (n < 0 || n > 65535)
        error("bytecode argument out of range",
              mkfix(n));
    string(cdr(Emitbuf))[a] = n >> 8;
    string(cdr(Emitbuf))[a+1] = n & 255;
}
```

The **Cts** variable will be bound to the *compile time stack* (*CTS*), which is implemented as a list, because not a lot of data will go through it. The CTS is used to store addresses of the arguments of forward jump and branch instructions. It is used for backpatching.

The **cpushval()** macro pushes a value to the CTS and the **cpopval()** function removes and returns the most recently pushed value. Values are packaged in fixnums, because unboxed integers (C **int**'s) cannot be stored in LISP9 lists. **cpopval()** signals an error, if the CTS is empty.

```
cell    Cts = NIL;
```

```
#define cpushval(x) (Cts = cons(mkfix(x), Cts))

cell cpopval(void) {
    cell    n;

    if (NIL == Cts)
        error("oops: compile stack underflow",
              UNDEF);
    n = car(Cts);
    Cts = cdr(Cts);
    return fixval(n);
}
```

The **swap()** function swaps the top two elements of the CTS or signals an error, if there are less than two elements on the stack.

```
void swap(void) {
    cell    x;

    if (NIL == Cts || NIL == cdr(Cts))
        error("oops: compile stack underflow",
              UNDEF);
    x = car(Cts);
    car(Cts) = cadr(Cts);
    cadr(Cts) = x;
}
```

All *primitive functions* of LISP9 will be *inlined* by the compiler. A primitive function of a language is a function that

- is an integral part of the language
- is pre-compiled, usually to machine code, for efficiency

In interpreted systems or systems using an abstract machine, primitive functions are typically implemented in the *implementation language* of the system and not in the language being implemented. Of course, both languages may be the same, and this is actually the case in many modern LISP implementations.

In LISP9, *primitive* functions are implemented in C, but not all *built-in* functions are. Pretty much all of the more abstract built-in functions are implemented in LISP9, because doing so in C would add a lot of complexity (see **appconv()**, for example) and the performance gain would be marginal.

For instance, **cons**, **+**, **rever**, **si=**, and **mkvec** are implemented as primitive function, but **mapcar**, **equal**, **foreach**, and **with-infile** are implemented in LISP9. In some cases, the decision is rather deliberate, and in some cases the implementation may change in future versions.

Inlining or *open-coding* a function means to insert the code for that function rather than emitting a call to that function. In the case of the LISP9 abstract machine, there is an individual *opcode* (or *instruction*) for each primitive function.

Traditionally, functions implemented in machine code are called *subroutines* or *SUBR*s. The term goes back all the way to the first LISP implementation [McCarthy1960b]. LISP9 has quite a few different kinds of SUBRs. See figure 44 for a summary.

Type	Description
SUBR0	nullary (zero-argument) subroutines
SUBR1	unary (single-argument) subroutines
SUBR2	binary (two-argument) subroutines
SUBR3	ternary (three-argument) subroutines
OSUBR0	nullary subroutines with an optional argument
OSUBR1	unary subroutines with an optional argument
LSUBR0	variadic (variable-argument) subroutines
LSUBR1	variadic subroutines with one mandatory argument

Fig. 44 – Types of subroutines

The SUBRn type expects exactly n arguments, OSUBRn expects either n or $n+1$ arguments, and LSUBRn expects at least n arguments, but also accepts any number greater than n.

LSUBRs also appear in [McCarthy1960b], but OSUBRs are probably an invention of the author of this book. They could easily be modeled using LSUBRs, but differentiating them as a separate type makes the life easier for the compiler.

The following functions map the names of primitive functions to their corresponding abstract machine opcodes. For instance, the **subr0()** function will return **OP_GENSYM** when given **P_gensym** (**gensym**). It will return -1 to indicate that the argument passed to it is not the name of any nullary primitive function (SUBR0).

There is one function for each type of subroutine listed in figure 44. All of them follow the same pattern, a series of statements of the form

if (x == P_*name***) return OP_***NAME***;**

where *name* is the name of a primitive function. A hash table could be chosen instead, but a sequence of "test-for-equality and branch" statements seems good enough for the small number of primitive routines in LISP9.

"Small" is subjective, though – feel free to skip ahead!

```
int subr0(cell x) {
    if (x == P_cmdline) return OP_CMDLINE;
    if (x == P_errport) return OP_ERRPORT;
    if (x == P_gc)      return OP_GC;
    if (x == P_gensym)  return OP_GENSYM;
    if (x == P_inport)  return OP_INPORT;
    if (x == P_obtab)   return OP_OBTAB;
    if (x == P_outport) return OP_OUTPORT;
    if (x == P_quit)    return OP_QUIT;
    if (x == P_symtab)  return OP_SYMTAB;
    return -1;
}

int subr1(cell x) {
    if (x == P_abs)     return OP_ABS;
    if (x == P_alphac)  return OP_ALPHAC;
    if (x == P_atom)    return OP_ATOM;
    if (x == P_caar)    return OP_CAAR;
    if (x == P_cadr)    return OP_CADR;
    if (x == P_car)     return OP_CAR;
    if (x == P_catchstar)
```

```
                            return OP_CATCHSTAR;
if (x == P_cdar)     return OP_CDAR;
if (x == P_cddr)     return OP_CDDR;
if (x == P_cdr)      return OP_CDR;
if (x == P_char)     return OP_CHAR;
if (x == P_charp)    return OP_CHARP;
if (x == P_charval) return OP_CHARVAL;
if (x == P_close_port)
                     return OP_CLOSE_PORT;
if (x == P_constp)   return OP_CONSTP;
if (x == P_ctagp)    return OP_CTAGP;
if (x == P_delete)   return OP_DELETE;
if (x == P_dump_image)
                     return OP_DUMP_IMAGE;
if (x == P_downcase)
                     return OP_DOWNCASE;
if (x == P_dump_image)
                     return OP_DUMP_IMAGE;
if (x == P_eofp)     return OP_EOFP;
if (x == P_eval)     return OP_EVAL;
if (x == P_existsp) return OP_EXISTSP;
if (x == P_fixp)     return OP_FIXP;
if (x == P_format)   return OP_FORMAT;
if (x == P_funp)     return OP_FUNP;
if (x == P_inportp) return OP_INPORTP;
if (x == P_liststr) return OP_LISTSTR;
if (x == P_listvec) return OP_LISTVEC;
if (x == P_load)     return OP_LOAD;
if (x == P_lowerc)   return OP_LOWERC;
if (x == P_mx)       return OP_MX;
if (x == P_mx1)      return OP_MX1;
if (x == P_not)      return OP_NULL;
if (x == P_null)     return OP_NULL;
if (x == P_numeric) return OP_NUMERIC;
if (x == P_open_infile)
                     return OP_OPEN_INFILE;
if (x == P_outportp)
                     return OP_OUTPORTP;
```

```
    if (x == P_pair)     return OP_PAIR;
    if (x == P_set_inport)
                         return OP_SET_INPORT;
    if (x == P_set_outport)
                         return OP_SET_OUTPORT;
    if (x == P_ssize)    return OP_SSIZE;
    if (x == P_stringp)  return OP_STRINGP;
    if (x == P_strlist)  return OP_STRLIST;
    if (x == P_symbol)   return OP_SYMBOL;
    if (x == P_symbolp)  return OP_SYMBOLP;
    if (x == P_symname)  return OP_SYMNAME;
    if (x == P_syscmd)   return OP_SYSCMD;
    if (x == P_untag)    return OP_UNTAG;
    if (x == P_upcase)   return OP_UPCASE;
    if (x == P_upperc)   return OP_UPPERC;
    if (x == P_veclist)  return OP_VECLIST;
    if (x == P_vectorp)  return OP_VECTORP;
    if (x == P_vsize)    return OP_VSIZE;
    if (x == P_whitec)   return OP_WHITEC;
    return -1;
}

int subr2(cell x) {
    if (x == P_cons)     return OP_CONS;
    if (x == P_div)      return OP_DIV;
    if (x == P_eq)       return OP_EQ;
    if (x == P_nreconc)  return OP_NRECONC;
    if (x == P_reconc)   return OP_RECONC;
    if (x == P_rem)      return OP_REM;
    if (x == P_sless)    return OP_SLESS;
    if (x == P_slteq)    return OP_SLTEQ;
    if (x == P_sequal)   return OP_SEQUAL;
    if (x == P_sgrtr)    return OP_SGRTR;
    if (x == P_sgteq)    return OP_SGTEQ;
    if (x == P_setcar)   return OP_SETCAR;
    if (x == P_setcdr)   return OP_SETCDR;
    if (x == P_sfill)    return OP_SFILL;
    if (x == P_siless)   return OP_SILESS;
```

```
    if (x == P_silteq)  return OP_SILTEQ;
    if (x == P_siequal) return OP_SIEQUAL;
    if (x == P_sigrtr)  return OP_SIGRTR;
    if (x == P_sigteq)  return OP_SIGTEQ;
    if (x == P_sref)    return OP_SREF;
    if (x == P_throwstar)
                        return OP_THROWSTAR;
    if (x == P_vfill)   return OP_VFILL;
    if (x == P_vref)    return OP_VREF;
    return -1;
}

int subr3(cell x) {
    if (x == P_sset)    return OP_SSET;
    if (x == P_substr)  return OP_SUBSTR;
    if (x == P_subvec)  return OP_SUBVEC;
    if (x == P_vset)    return OP_VSET;
    return -1;
}

int osubr0(cell x) {
    if (x == P_peekc)   return OP_PEEKC;
    if (x == P_read)    return OP_READ;
    if (x == P_readc)   return OP_READC;
    return -1;
}

int osubr1(cell x) {
    if (x == P_error)   return OP_ERROR;
    if (x == P_mkstr)   return OP_MKSTR;
    if (x == P_mkvec)   return OP_MKVEC;
    if (x == P_numstr)  return OP_NUMSTR;
    if (x == P_open_outfile)
                        return OP_OPEN_OUTFILE;
    if (x == P_prin)    return OP_PRIN;
    if (x == P_princ)   return OP_PRINC;
    if (x == P_strnum)  return OP_STRNUM;
    if (x == P_writec)  return OP_WRITEC;
```

```
    return -1;
}

int lsubr0(cell x) {
    if (x == P_times)   return OP_TIMES;
    if (x == P_plus)    return OP_PLUS;
    if (x == P_conc)    return OP_CONC;
    if (x == P_nconc)   return OP_NCONC;
    if (x == P_sconc)   return OP_SCONC;
    if (x == P_vconc)   return OP_VCONC;
    return -1;
}

int lsubr1(cell x) {
    if (x == P_bitop)   return OP_BITOP;
    if (x == P_max)     return OP_MAX;
    if (x == P_min)     return OP_MIN;
    if (x == P_minus)   return OP_MINUS;
    if (x == P_less)    return OP_LESS;
    if (x == P_lteq)    return OP_LTEQ;
    if (x == P_equal)   return OP_EQUAL;
    if (x == P_grtr)    return OP_GRTR;
    if (x == P_gteq)    return OP_GTEQ;
    if (x == P_cless)   return OP_CLESS;
    if (x == P_clteq)   return OP_CLTEQ;
    if (x == P_cequal)  return OP_CEQUAL;
    if (x == P_cgrtr)   return OP_CGRTR;
    if (x == P_cgteq)   return OP_CGTEQ;
    return -1;
}
```

The **subrp()** predicate returns truth, if its argument x is the name of *any* type of primitive function.

```
int subrp(cell x) {
    return subr0(x) >= 0 ||
           subr1(x) >= 0 ||
           subr2(x) >= 0 ||
```

```
            subr3(x) >= 0 ||
            osubr0(x) >= 0 ||
            osubr1(x) >= 0 ||
            lsubr0(x) >= 0 ||
            lsubr1(x) >= 0;
}
```

The **compexpr()** function, which is forward-declared here, will be used to compile all kinds of expressions, including special forms. The argument x is the expression to compile and t indicates whether the context of the expression is a *tail application context* (or just *tail context*), i.e. it is "true" when x is compiled in a place where a *function application* would be a *tail application*.

The following is an exhaustive list of all tail application contexts (indicated by "tail") in LISP9:

```
(if expr tail tail)
(if* expr tail)
(prog expr ... tail)
(lambda <formals> expr ... tail)
```

From this follows that the following contexts in derived syntax constructs are also tail application contexts:

```
(and expr ... tail)
(or expr ... tail)
(cond (expr ... tail) ...)
(cond (expr => tail) ...)
(case expr (list expr ... tail) ...)
(do <bindings> (expr ... tail) <body>)
```

The **compprog()** function compiles the **prog** special form. It simply compiles each expression of **prog** (except for the last one) in a non-tail context and the last one in a tail context, but *only if* **prog** itself is in a tail context.

In the following example, **prog** itself would be in a non-tail context and hence its last expression, b, would also be in a non-tail context:

```
(if (prog a b) c d)
```

Formally, **(prog** $x_1 \cdots x_n$**)** compiles to

$x_1; \cdots; x_n$

When there are no expressions in **prog**, code for loading **nil** is emitted, i.e. **(prog)** compiles to

quote **nil**

```
void compexpr(cell x, int t);

void compprog(cell x, int t) {
    x = cdr(x);
    if (NIL == x) {
        emitq(NIL);
        return;
    }
    while (cdr(x) != NIL) {
        compexpr(car(x), 0);
        x = cdr(x);
    }
    compexpr(car(x), t);
}
```

The **compsetq()** function compiles the **setq** special form. The form already has been closure-converted at this point and so **compsetq()** compiles **(setq (**%*arg n***)** *x***)** to

x; *setarg n*

and **(setq (**%*ref n v***)** *x***)** to

x; *setref n*

Note that arguments of LAM instructions will not print as separate bytes in the following. E.g., *setref n* is short for

setref (*n*/256) (*n mod* 256)

```
void compsetq(cell x) {
    compexpr(caddr(x), 0);
    if (caadr(x) == I_ref) {
```

```
        emitop(OP_SETREF);
        emitarg(fixval(cadadr(x)));
    }
    else if (caadr(x) == I_arg) {
        emitop(OP_SETARG);
        emitarg(fixval(cadadr(x)));
    }
    else {
        error("oops: unknown location in setq",
              x)
    }
}
```

The **compif()** function compiles the **if** and **if*** special forms. It will compile the latter, if the *star* argument is true. Formally, these forms compile as follows:

(if p c a)	(if p c)	(if* p a)
p;	*p*;	*p*;
brf 1;	*brf* 1;	*brt* 1;
c;	*c*;	*a*;
jmp 2;	1 ···	1 ···
1 *a*;		
2 ···		

where **1** and **2** symbolize addresses. The actual addresses are offsets in **Emitbuf** (and later in **Prog**). They are copied from the **Here** pointer at compile time.

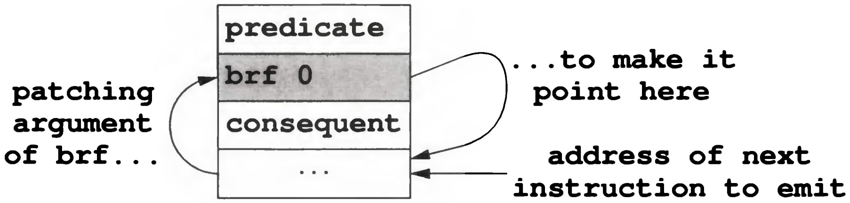


Fig. 45 – Backpatching a forward jump

Because the destinations of the branch and jump instructions are not known when they are emitted by **compif()**, the function will emit 0 in the place of the actual address, push the address of the

0 to the *compile time stack* (CTS), and then keep emitting code until it reaches the destination of the jump. At that point, it removes the address stored on the CTS and replaces the 0 with the actual address. This process is known as *backpatching*. See figure 45.

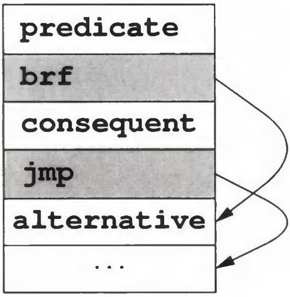


Fig. 46 – Jump around jump

In the case of a two-branched **if**, there are two addresses on the CTS, which have to be swapped at some point, because the paths of the jumps cross. This kind of machine code is called a *jump around jump*. It is depicted in figure 46.

```
void compif(cell x, int t, int star) {
    compexpr(cadr(x), 0);
    emitop(star? OP_BRT: OP_BRF);
    cpushval(Here);
    emitarg(0);
    compexpr(caddr(x), t);
    if (cdddr(x) != NIL) {
        emitop(OP_JMP);
        cpushval(Here);
        emitarg(0);
        swap();
        patch(cpopval(), Here);
        compexpr(cadddr(x), t);
    }
    patch(cpopval(), Here);
}
```

The **setupenv()** function expects an *initmap* m and emits

instructions for populating a *local environment*. It emits

cparg k n

for each initmap entry of the form **(a** *k n y***)** and

cpref k n

for each entry of the form **(e** *k n y***)**

The *y* part of each initmap entry (the reference to the symbol table) is ignored by this function.

```
void setupenv(cell m) {
    while (m != NIL) {
        if (caar(m) == I_e)
            emitop(OP_CPREF);
        else if (caar(m) == I_a)
            emitop(OP_CPARG);
        else
            error(
              "oops: unknown location in closure",
              m);
        emitarg(fixval(cadar(m)));
        emitarg(fixval(caddar(m)));
        m = cdr(m);
    }
}
```

The **dottedp()** predicate returns truth, if *x* is a dotted argument list (which includes pairs and symbols).

```
cell dottedp(cell x) {
    while (pairp(x)) x = cdr(x);
    return x != NIL;
}
```

The **compcls()** function compiles a closure, i.e. an S-expression generated by the **clsconv()** function:

(**%closure** *formals initmap expression* $\cdots$)

The above is compiled to an abstract machine code fragment of the following form:

	jmp	1		jump around closure body
2	*enter*	*na*		*entcol*, if the closure is variadic
	expression			
	...			
	return			
1	*mkenv*	$\lvert m\rvert$		*propenv*, if $\lvert m\rvert = 0$
	cp...	x_0	0	
	...			
	cp...	$x_{\lvert m\rvert-1}$	$\lvert m\rvert - 1$	
	closure	2		entry point is 2

Here *na* is the number of formal arguments (the flat length of *formals*) and $\lvert m\rvert$ is the length of *initmap*. The generated code jumps around the body fo the closure, which begins at *enter* or *entcol* and ends with a *return* instruction. The code following the body sets up the local environment of the closure: *mkenv* places a fresh vector of the length $\lvert m\rvert$ in *Acc* and the following *cparg* and *cpref* instructions copy bindings to the vector. The final *closure* instruction creates a closure with environment *Acc* and the given entry point.

```
void compcls(cell x) {
    int     a, na;
    cell    b, m;

    emitop(OP_JMP);
    cpushval(Here);
    emitarg(0);
    a = Here;
    na = length(flatargs(cadr(x)));
    if (dottedp(cadr(x))) {
        emitop(OP_ENTCOL);
        emitarg(na-1);
    }
    else {
```

```
        emitop(OP_ENTER);
        emitarg(na);
    }
    b = cons(S_prog, cdddr(x));
    protect(b);
    compexpr(b, 1);
    unprot(1);
    emitop(OP_RETURN);
    patch(cpopval(), Here);
    m = caddr(x);
    if (m != NIL) {
        emitop(OP_MKENV);
        emitarg(length(m));
        setupenv(m);
    }
    else {
        emitop(OP_PROPENV);
    }
    emitop(OP_CLOSURE);
    emitarg(a);
}
```

The **compapply()** function compiles the **apply** special form. It first transforms the more general form

(apply *fn* $x_1 \cdots x_n$ *list***)**

to the simpler form

(apply *fn* *list***)**

by emitting code that performs the following cons operations at run time:

(apply *fn* **(cons** x_1 **(cons** $\cdots$ **(cons** x_n *list***))))**

After performing the cons operations, the emitted code is basically

consed-list; *fn*; *applis*

When the application takes place in a tail context, *applist* is emitted in the place of *applis*.

```
void compapply(cell x, int t) {
    cell    xs;

    xs = reverse(cddr(x));
    protect(xs);
    compexpr(car(xs), 0);
    for (xs = cdr(xs); xs != NIL; xs = cdr(xs)) {
        emitop(OP_PUSH);
        compexpr(car(xs), 0);
        emitop(OP_CONS);
    }
    emitop(OP_PUSH);
    unprot(1);
    compexpr(cadr(x), 0);
    emitop(t? OP_APPLIST: OP_APPLIS);
}
```

The **compapp()** function compiles a *function application*, where "function application" means the application of a *closure* in this context. Most of the **compapp()** function deals with building the argument part of a *call frame* as discussed earlier in this chapter. Arguments are passed to closures in reverse (*right-to-left*) order and the total number of arguments is passed on top of the argument section of the stack frame. So, generally

$(fn\ \ x_1\ \cdots\ x_n)$

compiles so

$x_1;\ \ push$
...
$x_n;\ \ push$
$pushval\ n$
$fn;\ \ apply$

When function application takes place in a tail context, a *tailapp* instruction is emitted in the place of *apply*.

```
void compapp(cell x, int t) {
```

```
    cell    xs;

    xs = reverse(cdr(x));
    protect(xs);
    while (xs != NIL) {
        compexpr(car(xs), 0);
        emitop(OP_PUSH);
        xs = cdr(xs);
    }
    unprot(1);
    emitop(OP_PUSHVAL);
    emitarg(length(cdr(x)));
    compexpr(car(x), 0);
    emitop(t? OP_TAILAPP: OP_APPLY);
}
```

The **compsubr0()** function compiles applications of *nullary (zero-argument) inline functions*, like **gensym** or **obtab** or **inport**. It merely makes sure that no arguments are passed to the function and emits the corresponding instruction *op* (as delivered by the **subr0()** function).

```
void compsubr0(cell x, int op) {
    ckargs(x, 0, 0);
    emitop(op);
}
```

The **compsubr1()** function compiles applications of *unary (single-argument) inline functions*, like **abs** or **cadr** or **catch***. These functions expect their single argument in the accumulator, so

(*fn* *x*)

compiles to

x; *op*

where *op* is the opcode corresponding to *fn*.

When *fn* equals **catch***, an additional *apply* instruction will be

emitted, because **catch*** *applies* its argument to a catch tag. Therefore **(catch*** *x***)** compiles to

x; *catch*; *apply*

However, the *catch* instruction *pushes* its result (*ctag*) to the stack instead of returning it in *Acc*, so the meaning of the above code is

ctag; *push*; *x*; *apply*

i.e., apply *x* to the catch tag returned by the *catch* instruction.

```
void compsubr1(cell x, int op) {
    ckargs(x, 1, 1);
    compexpr(cadr(x), 0);
    emitop(op);
    if (OP_CATCHSTAR == op) emitop(OP_APPLY);
}
```

The **compsubr2()** function compiles applications of *binary (two-argument) inline functions*, like **cons** or **eq** or **div**. Binary inline functions expect their second argument on the stack and the first one in the accumulator, so

(*fn* *x* *y*)

compiles to

y; *push*; *x*; *op*

There are no exceptions in this case.

```
void compsubr2(cell x, int op) {
    ckargs(x, 2, 2);
    compexpr(caddr(x), 0);
    emitop(OP_PUSH);
    compexpr(cadr(x), 0);
    emitop(op);
}
```

The **compsubr3()** function compiles applications of *ternary (three-argument) inline functions*, like **substr** or **subvec** or **vset**. Ternary

inline functions expect their second and third argument on the stack and the first one in the accumulator, so

(*fn x y z***)**

compiles to

z; *push*; *y*; *push*; *x*; *op*

There are no exceptions in this case and there are no inline functions with more than three arguments.

```
void compsubr3(cell x, int op) {
    ckargs(x, 3, 3);
    compexpr(cadddr(x), 0);
    emitop(OP_PUSH);
    compexpr(caddr(x), 0);
    emitop(OP_PUSH);
    compexpr(cadr(x), 0);
    emitop(op);
}
```

The **composubr0()** function compiles applications of inline functions of the type *OSUBR0*, i.e. inline functions with zero arguments and one optional argument, like **read**, **readc**, and **peekc**.

Because there are no variadic instructions, the compiler will supply a default argument when no optional argument is given. In the case of OSUBR0 functions, this default argument is always the current input port. Hence, for instance, **(readc)** compiles to

inport; *readc*

When an optional argument *o* is given, the compiler will emit code to evaluate the optional argument instead:

o; *readc*

```
void composubr0(cell x, int op) {
    ckargs(x, 0, 1);
    if (NIL == cdr(x))
```

```
        emitop(OP_INPORT);
    else
        compexpr(cadr(x), 0);
    emitop(op);
}
```

The following variables will be bound to some objects that are used as default arguments in OSUBR-type functions. **Blank** will be bound to **#\sp**, **Zero** to the fixnum 0, **One** to the fixnum 1, and **Ten** to the fixnum 10.

```
cell    Blank = NIL;
cell    Zero = NIL;
cell    One = NIL;
cell    Ten = NIL;
```

The **composubr1()** function compiles applications of inline functions of the type *OSUBR1*, i.e. inline functions with one argument and one optional argument, like **prin**, **mkvec**, or **numstr**. This function is basically a cross between **compsubr1()** and **composubr0()**.

Because different OSUBR1 functions have different default arguments, though, the proper default has to be supplied for each function. For instance, **mkstr** expects the blank character **#\sp** as its default, **mkvec** expects **nil**, and **prin** expects the current output port. Formally, **composubr1()** compiles

(*fn* *x*)

to

default; *push*; *x*; *op*

where *default* is the default argument, *x* is the supplied argument, and *op* is the opcode (instruction) corresponding to *fn*. When an optional argument is supplied, it replaces *default*.

A special case is **error**, which does not have any default. In this case, there are two different instructions, *error* for signaling an error without an additional argument and *error2*, which does

expect an additional argument. When no optional argument is passed to **error**, the following code will be emitted:

push; *x*; *error*

where *x* is the supplied argument. The *push* instruction will push whatever is in the accumulator at that point, leaving a spurious value on the stack, but this does not matter, because the *error* instruction will never return and the context containing the spurious value will be abandoned.

```
void composubr1(cell x, int op) {
    ckargs(x, 1, 2);
    if (NIL == cddr(x)) {
        if (OP_ERROR == op) {
            /**/
        }
        else if (OP_MKSTR == op) {
            emitq(Blank);
        }
        else if (OP_MKVEC == op) {
            emitq(NIL);
        }
        else if (OP_OPEN_OUTFILE == op) {
            emitq(NIL);
        }
        else if (OP_NUMSTR == op ||
                 OP_STRNUM == op)
        {
            emitq(Ten);
        }
        else if (OP_WRITEC == op ||
             OP_PRIN == op ||
             OP_PRINC == op)
        {
            emitop(OP_OUTPORT);
        }
    }
    else {
```

```
        if (OP_ERROR == op) op = OP_ERROR2;
        compexpr(caddr(x), 0);
    }
    emitop(OP_PUSH);
    compexpr(cadr(x), 0);
    emitop(op);
}
```

The **complsubr0()** function compiles applications of inline functions of the type *LSUBR0*, i.e. inline functions with any number of arguments, like **+**, **conc**, or *****.

When no arguments at all are passed to these functions, their applications will compile to a default value that is specific to the function. For instance, **(+)** will emit the fixnum 0, **(*)** will emit the fixnum 1, **(conc)** will emit **nil**, and **(sconc)** will emit an empty string.

When a single argument is passed to these functions, the compiler will just emit code to evaluate that argument. E.g.

```
(* (+ 2 3))
```

will emit code to compute **(+ 2 3)**. Note that this implies that the wrong type of object can be passed as a single argument to LSUBR0 functions. For instance, the code generated for **(+ 'foo)** will happily evaluate to **foo** rather than signaling an error. This is a bug! Fixing it is left as an interesting exercise!

When there is more than one argument, there are two ways to generate code. For the **+** and *****, operations, the following code is emitted (where op denotes the operation and x_i the n arguments):

$$x_1;\ push;\ x_2;\ op;\ push;\ x_3;\ op;\ \cdots\ push;\ x_n;\ op$$

Because arguments are passed in right-to-left order, the code reproduces the application

$$(op\ \ x_n\ \ \cdots\ \ x_2\ \ x_1)$$

but since the $+$ and $\times$ operators are commutative, it does not matter. For the concatenation functions, the following code is generated:

nil; *push*; x_n; *cons*; *push*; x_{n-1}; *cons* $\cdots$ *push*; x_1; *cons*; *op*

This code resembles the application

$$(op\ (cons\ x_1\ \cdots\ (cons\ x_{n-1}\ (cons\ x_n\ \textbf{nil}))\ \cdots\ ))$$

I.e., it first creates a list of all of its arguments and then passes it to the desired operator. Of course, the operator *op* itself could take the place of *cons*, but in that case $n-1$ concatenation operations would have to be performed (and $n-1$ objects allocated), while a microcode function concatenating multiple objects at the same time can be implemented much more efficiently.

For example, a function appending n strings can allocate one string of the required length and then copy the characters of all source strings to the destination. This approach is a simple yet effective optimization.

```
void complsubr0(cell x, int op) {
    if (NIL == cdr(x)) {
        if (OP_PLUS == op)
            emitq(Zero);
        else if (OP_TIMES == op)
            emitq(One);
        else if (OP_VCONC == op)
            emitq(Nullvec);
        else if (OP_SCONC == op)
            emitq(Nullstr);
        else if (OP_CONC == op)
            emitq(NIL);
        else if (OP_NCONC == op)
            emitq(NIL);
    }
    else if (NIL == cddr(x)) {
        compexpr(cadr(x), 0);
    }
    else if (OP_CONC == op || OP_SCONC == op ||
             OP_VCONC == op || OP_NCONC == op)
    {
        x = reverse(cdr(x));
```

```
        protect(x);
        emitq(NIL);
        while (x != NIL) {
            emitop(OP_PUSH);
            compexpr(car(x), 0);
            emitop(OP_CONS);
            x = cdr(x);
        }
        unprot(1);
        emitop(op);
    }
    else {
        x = cdr(x);
        protect(x);
        compexpr(car(x), 0);
        x = cdr(x);
        while (x != NIL) {
            emitop(OP_PUSH);
            compexpr(car(x), 0);
            emitop(op);
            x = cdr(x);
        }
        unprot(1);
    }
}
```

The **compbitop()** function compiles applications of the *bit operator* **bitop**, which implements 18 different bit operations, which are outlined in figure 59 on page 457 in the appendix.

The first argument of **bitop** evaluates to a fixnum indicating the operation. It is compiled first and pushed to the stack where it will remain up to the end of the operation. Bit operations are carried out as follows, where *op* denotes the bit operation specified in x_1, the first argument of **bitop**:

x_1; *push*; x_2; *push*; x_3; *op*; $\cdots$ *push*; x_n; *op*; *drop*

Since the *drop* instruction removes an element from the top of the stack without touching *Acc*, the result of the program will be left in

the accumulator.

Note that instructions will be generated in the same way as for the + and * operations, but the **bitop** microcode will swap its arguments, so the operands are combined in the proper order.

```
void compbitop(cell x) {
    if (NIL == cddr(x) || NIL == cdddr(x))
        error("bitop: too few arguments",
              cdr(x));
    compexpr(cadr(x), 0);
    emitop(OP_PUSH);
    x = cddr(x);
    compexpr(car(x), 0);
    for (x = cdr(x); x != NIL; x = cdr(x)) {
        emitop(OP_PUSH);
        compexpr(car(x), 0);
        emitop(OP_BITOP);
    }
    emitop(OP_DROP);
}
```

The **complsubr1()** function compiles applications of inline functions of the type *LSUBR1*, i.e. inline functions with at least one, but otherwise any number of arguments, like **–**, **max**, or **<**.

Applications of **bitop** are delegated to **compbitop()**, above. When there is only one single argument, **min** and **max** return that argument and the *comparison predicates*, like **=**, **c<**, etc, return truth, because any set of one element trivially satisfies equivalence well as any given ordering predicate. E.g. all elements of the set **{5}** are equal, and the set is in (strict) monotonically ascending and (strict) monotonically descending order.

When only a single argument is passed to **–**, a *negate* operation will be generated:

x; *negate*

In the general case with more than one argument, the **–**, **min**, and **max** operations compile like **+** and *, but the microcode of the

minus operation swaps its arguments, so that its operands are ordered properly.

The comparison predicates compile exactly like **–**, but in addition they keep their intermediate result on the stack. This intermadiate result defaults to "truth" and when the microcode of one of these operations detects a pair of operands that does not satisfy their associated condition, they change the top of the stack to **nil**. Generally, comparison predicates *op* compile as follows:

$$pushtrue;\; x_1;\; push;\; x_2;\; op;\; \cdots\; push;\; x_n;\; op;\; pop$$

The *pop* operation at the end of the code removes the intermediate result from the stack and places it in the accumulator.

```
void complsubr1(cell x, int op) {
    ckargs(x, 1, -1);
    if (OP_BITOP == op) {
        compbitop(x);
        return;
    }
    if (NIL == cddr(x)) {
        if (OP_MIN == op || OP_MAX == op) {
            compexpr(cadr(x), 0);
        }
        else if (OP_MINUS == op) {
            compexpr(cadr(x), 0);
            emitop(OP_NEGATE);
        }
        else {
            emitq(TRUE);
        }
    }
    else {
        if (op != OP_MINUS &&
            op != OP_MIN &&
            op != OP_MAX)
        {
            emitop(OP_PUSHTRUE);
        }
```

```
        x = cdr(x);
        compexpr(car(x), 0);
        for (x = cdr(x); x != NIL; x = cdr(x)) {
            emitop(OP_PUSH);
            compexpr(car(x), 0);
            emitop(op);
        }
        if (op != OP_MINUS &&
            op != OP_MIN &&
            op != OP_MAX
        )
            emitop(OP_POP);
    }
}
```

The **compexpr()** function compiles the expression x. When $t \neq 0$, compilation takes place in a tail application context. Compilation of most forms is delegated to one of the above functions, but the following cases are handled by **compexpr()** itself:

Atoms compile to literal objects.

Arguments of the form *(%arg n)* compile to *arg* instructions and references of the form *(%ref n m)* compile to *ref* instructions. The second argument of *ref*, which still is a symbol at this point, compiles to a symbol table slot. When the symbol is *uninterned* (i.e. generated by **gensym**), then a reference to slot 0 is generated, which contains a symbol named "**?**". This name will be used in the place of uninterned symbols in reference traces in error reporting.

Applications of **macro** compile to *macro* instructions with a symbol table slot as argument. In this case, the symbol *must* be interned, or an error will be signalled.

Comments compile to an empty program fragment. Hence they are not allowed in any context where a value *must* be present. Otherwise, what would

```
(if x (--) y)
```

evaluate to? (Seriously, do think about it! Would it be a good idea? Why? Why not?)

```
void compexpr(cell x, int t) {
    int     op;
    cell    y;

    if (atomp(x)) {
        emitq(x);
    }
    else if (car(x) == S_quote) {
        emitq(cadr(x));
    }
    else if (car(x) == I_arg) {
        emitop(OP_ARG);
        emitarg(fixval(cadr(x)));
    }
    else if (car(x) == I_ref) {
        emitop(OP_REF);
        emitarg(fixval(cadr(x)));
        y = htlookup(Symhash, caddr(x));
        if (UNDEF == y)
            emitarg(0);
        else
            emitarg(fixval(cdr(y)));
    }
    else if (car(x) == S_if) {
        compif(x, t, 0);
    }
    else if (car(x) == S_ifstar) {
        compif(x, t, 1);
    }
    else if (car(x) == I_closure) {
        compcls(x);
    }
    else if (car(x) == S_prog) {
        compprog(x, t);
    }
```

```
    else if (car(x) == S_setq) {
        compsetq(x);
    }
    else if (car(x) == S_apply) {
        compapply(x, t);
    }
    else if (car(x) == S_macro) {
        compexpr(caddr(x), 0);
        emitop(OP_MACRO);
        y = htlookup(Symhash, cadr(x));
        if (UNDEF == y)
            error("oops: unknown name in MACRO",
                  cadr(x));
        emitarg(fixval(cdr(y)));
    }
    else if (car(x) == S_comment) {
        /**/
    }
    else if ((op = subr0(car(x))) >= 0) {
        compsubr0(x, op);
    }
    else if ((op = subr1(car(x))) >= 0) {
        compsubr1(x, op);
    }
    else if ((op = subr2(car(x))) >= 0) {
        compsubr2(x, op);
    }
    else if ((op = subr3(car(x))) >= 0) {
        compsubr3(x, op);
    }
    else if ((op = osubr0(car(x))) >= 0) {
        composubr0(x, op);
    }
    else if ((op = osubr1(car(x))) >= 0) {
        composubr1(x, op);
    }
    else if ((op = lsubr0(car(x))) >= 0) {
        complsubr0(x, op);
```

```
    }
    else if ((op = lsubr1(car(x))) >= 0) {
        complsubr1(x, op);
    }
    else { /* application */
        compapp(x, t);
    }
}
```

The **subprog()** function is basically a simplified version of **substr**, which extracts bytes from a string and returns another string containing the extracted bytes. **subprog()** extracts the bytes from positions 0 to $k-1$.

```
cell subprog(cell x, int k) {
    cell    n;
    byte    *sx, *sn;
    int     i, j;

    n = mkstr(NULL, k);
    sx = string(x);
    sn = string(n);
    j = 0;
    for (i=0; i<k; i++) {
        sn[j] = sx[i];
        j++;
    }
    return n;
}
```

The **compile()** function compiles the expression x and returns the resulting LISP9 abstract machine program. It allocates the initial *emit buffer* (**Emitbuf**), clears the compile time stack, and resets the **Here** pointer to 0. Then it compiles the expression x and appends a *halt* instruction. Finally, it creates another *bytecode container* that is just big enough to hold the resulting LAM program and copies the program to that container. It "forgets" the emit

buffer by setting it to **nil** and returns the new container.

compile() is the complete interface to the compiler. It expects a closure-converted expression x and returns a ready-to-run LAM program.

```
cell compile(cell x) {
    cell    n;

    Emitbuf = mkatom(T_BYTECODE,
                     mkstr(NULL, CHUNKSIZE));
    Here = 0;
    Cts = NIL;
    compexpr(x, 0);
    emitop(OP_HALT);
    n = mkatom(T_BYTECODE, subprog(cdr(Emitbuf),
                                   Here));
    Emitbuf = NIL;
    return n;
}
```

15. Macro Expansion

Macro expansion is the process that translates an expression containing any *derived special forms* to an expression containing only *primitive forms*. The primitive *special* forms of LISP9 are

```
apply     if*       prog
def       lambda    quote
if        macro     setq
```

All other special forms, such as **let**, **labels**, **cond**, or **do**, are derived special forms, because they combine other special forms and expressions to implement their semantics. For instance, the derived special form

```
(let ((v a) ...) x ...)
```

expands to

```
((lambda (v ...) x ...) a ...)
```

This is the simplest case of macro expansion, where a derived special form is passed to the macro expander and the expander immediately delivers the (primitive) expanded form.

Things are not always that simple, though. For instance, the loop expression

```
(do ((v i s)) (t x1 ...) x2 ...)
```

expands in multiple steps (where → means "expands to"):

```
(do ((v i s))
    (t x1 ...)
  x2 ...)

→ (labels
    ((G1 (lambda (v)
           (if t (prog x1 ...)
                 (prog x2 ... (G1 s))))))
    (G1 i))

→ (let ((G1 nil))
```

```
    (setq G1 (lambda (v)
               (if t (prog x1 ...)
                     (prog x2 ... (G1 s)))))
    (G1 i))

→ ((lambda (G1)
    (setq G1 (lambda (v)
               (if t (prog x1 ...)
                     (prog x2 ... (G1 s)))))
    (G1 i))
   nil)
```

Then, of course, derived special forms do not always have to be located right at the beginning of a form. They can be buried deeply inside of other expressions.

In order to transform expressions to their expanded forms, the macro expander has to

- locate derived special forms
- transform them to their primitive forms
- replace the derived forms with primitive forms

Locating all derived special forms, or *macro applications*, inside of an expression is as simple as mapping the expander recursively over the expression x, i.e.:

```
(mapcar expand x)
```

When this is done *inside of the expander*, all subexpressions will be expanded recursively. Even replacing them is just a matter of making the “expand” function return the proper value, namely the expanded form.

The hard part is to actually *transform* the macro application to a primitive form. LISP9, like many LISPs since the early days, uses *low-level macro transformers*, which are LISP functions that expect a (possibly destructured) derived special form as their argument(s) and return another form, which may or may not be primitive.

For instance, the **do** transformer used in the expansion above returns a **labels** form and therefore its result is not primitive and

has to be expanded further.

Macro expansion is part of a larger process called *evaluation*. This process reduces any given well-formed LISP expression to its *value* (sometimes also called *normal form*, if the expression is purely functional). The process is depicted in figure 47. The **eval** function, which performs evaluation, is indicated by a gray box with a dashed outline. An S-expression (a LISP program) is passed to **eval**, and it returns another S-expression, which is the value of the program.

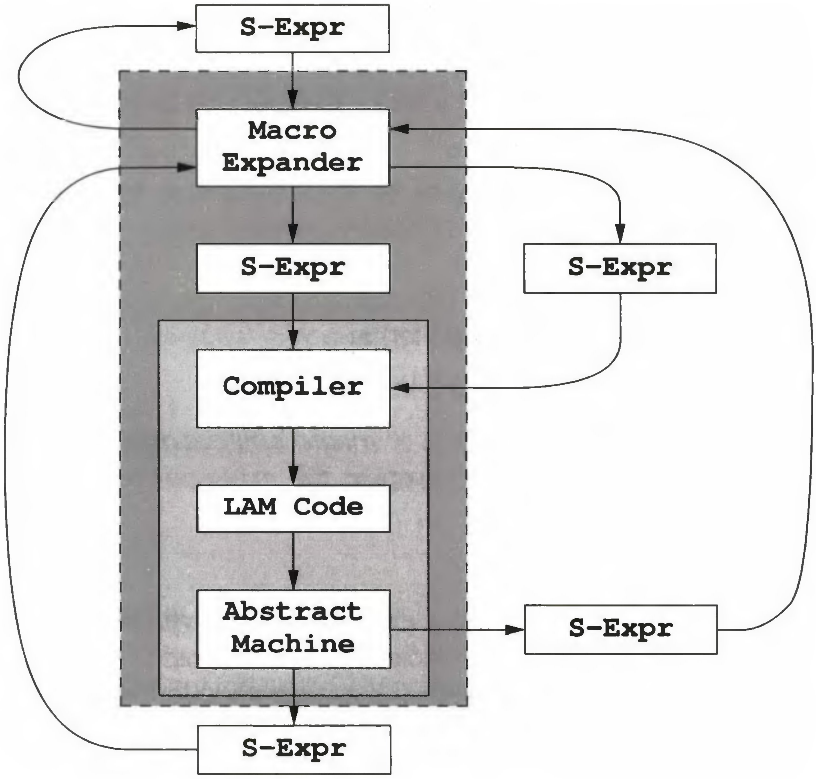


Fig. 47 – The process of macro expansion, two models

When ignoring the specifics of macro expansion for a moment, the process of evaluation is very straightforward, as indicated by the straight downward arrows in figure 47. The input expression is macro-expanded, compiled, and interpreted, and the result drops out at the end.

However, in order to macro-expand an expression, an application

of the corresponding transformer has to be created, compiled, and interpreted, and the macro expander has to perform these steps *before* compilation and interpretation of the (expanded) input expression takes places.

This is indicated by the right hand side arrows in the figure: The macro expander submits the transformer application for compilation and interpretation (inner gray box) and collects the result, which is an expanded form. It then passes that expanded form on to the compiler for further evaluation.

The inner gray box in the figure accounts for a large part of **eval**, though, so would it not make sense to just use **eval** for compiling and interpreting the transformer application, as indicated by the left hand side arrows in the figure? At a first glance, this may look problematic, because the first thing **eval** does it to expand its argument, which is exactly what the macro expander tried to achieve in the first place, so infinite recursion would take place. However, the argument of any macro transformer M has to be *quoted* anyway, or it would evaluate before the transformer would have an opportunity to transform it:

```
(apply M (quote <derived special form>))
```

An application of **quote** is a primitive special form, so it will never be expanded, and hence applications of macro transformers can simply be passed to **eval** for evaluation. Of course, this approach requires **eval** to be *reentrant*, but this is merely an implementation detail that will be discussed later.

There are alternative systems for defining derived special forms that do not require access to the full LISP language at macro expansion time. Such systems are usually based on pattern matching and substitution of variables in templates. See [CR1991] or [R4Rs] for an example of such a system. Combined systems, using both pattern matching and LISP code, also exist [DHB1992].

The variable **Macros** will be bound to a deep-binding environment that maps macro names to macro transformers.

```
cell    Macros = NIL;
```

The **newmacro()** function assigns the macro transformer *fn*, which must be a compiled function, to the symbol in symbol table slot *id*. When a binding for *id* already exists in **Macros**, its value will be changed to *fn*. When no binding for *id* exists, a fresh one will be created and added to the environment.

Note that symbols are directly bound to *values* in the **Macros** environment. There is no need for an additional location layer.

```
void newmacro(int id, cell fn) {
    cell    n, name;

    if (!closurep(fn))
        expect("macro", "closure", fn);
    name = vector(Symbols)[id];
    n = assq(name, Macros);
    if (NIL == n) {
        n = cons(name, fn);
        Macros = cons(n, Macros);
    }
    else {
        cdr(n) = fn;
    }
}
```

The **expand()** function, which is forward-declared here, will expand an expression *x* to its primitive form. When the argument *r* is true, expansion will be recursive, otherwise it will be non-recursive. Non-recursive expansion stops as soon as the first derived special form in an expression has been transformed.

```
cell expand(cell x, int r);
```

The **mapexp()** function maps **expand()** over *x*, i.e. it does

(map (lambda (x) (expand x r)) *x***)**

The function GC-protects its argument, *x*, because it may be an

intermediate result produced by an earlier expansion step.

```
cell mapexp(cell x, int r) {
    cell    p, n, new;

    protect(x);
    protect(n = NIL);
    p = x;
    while (pairp(p)) {
        new = expand(car(p), r);
        n = cons(new, n);
        car(Protected) = n;
        p = cdr(p);
    }
    if (p != NIL)
        error("dotted list in program", x);
    n = nreverse(unprot(1));
    unprot(1);
    return n;
}
```

The **zip()** function "zips up" the lists a and b to create a list of pairs, i.e.:

(zip '(a_1 $\cdots$ a_n) '(b_1 $\cdots$ b_n))
  => ((a_1 . b_1) $\cdots$ (a_n . b_n))

It will be used to transform local **def** and **defun** forms to **labels** forms.

```
cell zip(cell a, cell b) {
    cell    n, p;

    protect(n = NIL);
    while (a != NIL && b != NIL) {
        p = cons(car(a), car(b));
        n = cons(p, n);
        car(Protected) = n;
```

```
        a = cdr(a);
        b = cdr(b);
    }
    unprot(1);
    return nreverse(n);
}
```

The LISP9 macro expander does not only transform derived special forms, but also some built-in special forms, such as **defun** and **defmac**.

The **expanddef()** function expands these forms. See below.

```
cell expanddef(cell x);
```

The **expandbody()** function collects local **def** and **defun** special forms in the function or macro body x and transforms them to a **labels** special form:

```
((def v1 a1)    -->    (labels ((v1 a1)
 ...                            ...
 (def vn an)                    (vn an))
 expr1                   expr1
 ...)                    ...)
```

If a **defun** form is found in the place of a **def** form, then it will be expanded before the above transformation takes place. E.g.

```
((defun (f x) x) ...)
-> ((def f (lambda (x) x)) ...)
-> (labels ((f (lambda (x) x))) ...)
```

When there are no local definitions in the body x, it will be returned unchanged.

Comments that are interspersed among local definition will be ignored by **expandbody()**.

```
cell expandbody(cell x) {
    cell    n, vs, as;
```

```
protect(vs = NIL);
protect(as = NIL);
while (pairp(x) &&
       pairp(car(x)) &&
       (caar(x) == S_def ||
        caar(x) == S_defun ||
        caar(x) == S_comment))
{
    if (caar(x) == S_comment) {
        /**/
    }
    else if (caar(x) == S_def) {
        n = car(x);
        vs = cons(cadr(n), vs);
        cadr(Protected) = vs;
        n = cons(caddr(n), NIL);
        as = cons(n, as);
        car(Protected) = as;
    }
    else {
        n = expanddef(car(x));
        protect(n);
        vs = cons(cadr(n), vs);
        caddr(Protected) = vs;
        n = cons(caddr(n), NIL);
        as = cons(n, as);
        cadr(Protected) = as;
        unprot(1);
    }
    x = cdr(x);
}
if (NIL == vs) {
    unprot(2);
    return x;
}
as = car(Protected) = nreverse(as);
vs = cadr(Protected) = nreverse(vs);
```

```
    n = cons(zip(vs, as), x);
    n = cons(S_labels, n);
    n = cons(n, NIL);
    unprot(2);
    return n;
}
```

The **expanddef()** function expands the **defun** and **defmac** special forms, which are built into the LISP9 system at a low level, i.e. they are not themselves implemented as derived special forms.

The following transformation is performed:

(defun (*fn* . *formals*) $expr_1$ ···**)**
→ **(def** *fn* **(lambda** *formals* $expr_1$ ···**)**

where *formals* denotes a formal argument list, which can be a (proper) list, a dotted list, or a single symbol. So, for instance,

```
(defun (f x) y)   →   (def f (lambda (x) y))
```

(because **(f x)** equals **(f . (x))**) and

```
(defun (f . x) y)   →   (def f (lambda x y))
```

The transformation is the same in the case of **defmac**, but it produces a **macro** special form instead of a **def** special form:

(defmac (*fn* . *formals*) $expr_1$ ···**)**
→ **(macro** *fn* **(lambda** *formals* $expr_1$ ···**)**

In either case, the body $expr_1$ ··· of the form is also transformed by converting any local definitions contained in it to a **labels** special form.

```
cell expanddef(cell x) {
    char    b[100];
    cell    n;

    if (!pairp(cadr(x))) {
        sprintf(b, "%s: expected signature",
                symname(car(x)));
        error(b, cadr(x));
```

```
    }
    n = cons(cdadr(x), expandbody(cddr(x)));
    n = cons(S_lambda, n);
    n = cons(n, NIL);
    n = cons(caadr(x), n);
    n = cons(car(x) == S_defun? S_def: S_macro,
             n);
    return n;
}
```

Macro expansion is a process that involves a lot of recursion. For instance, a **let*** expression with n clauses recurses $3n$ times, because each clause is first converted to nested **let**, then to **lambda**, and then to itself.

It is easy to conceive macros that would recurse even more frequently, for example derived special forms that implement modules, where any number of definitions can be contained in a module and each definition may contain applications of other macros, like **let***.

On the other hand, erroneous macros may recurse indefinitely, like

```
(defmac (wrong x) @(wrong ,x))
```

thereby causing the LISP system to run out of memory. Depending on the speed of the abstract machine and the amount of memory available to the system, this can take a lot of time and allocate a lot of resources.

So there has to be a way to limit recursion. The easiest way to do this is to set an arbitrary limit on the number of times that a special from can be transformed. The variable **Mxlev** ("macro expansion level") keeps track of the number of recursive macro expansions. When it exceeds a certain limit (**MXMAX**), an error will be signalled. **Mxlev** is declared "volatile", because it may be altered asynchronously to -1, thereby indicating that a computation was aborted by the user.

```
volatile cell   Mxlev = 0;
```

As discussed initially in this chapter, the macro expander will use **eval** to evaluate applications of macro transformers. The argument r of **eval** is set to 1, when **eval** is called recursively.

```
cell eval(cell x, int r);
```

The **expand()** function macro-expands the expression x. When the argument r is true, it will expand the form to its primitive form and otherwise, it will just expand the leftmost derived special form in x. In either case it returns the expanded form.

The function will also expand **defun** and **defmac** special forms. The effect of the r argument on the expansion of those forms is unspecified.

There are 7 different cases to distinguish in **expand()**:

(1) When x is an atom, it is returned immediately.

(2) When x is an application of **quote**, it is returned immediately.

(3) When x is a *lambda function*, then the *body* of the function will be expanded and the **lambda** keyword and the formal argument list will be left unchanged.

(4) When x is a **defun** special form, then it will be transformed to a **def** special form and subsequently expanded again.

(5) When x is a **defmac** special form, then it will be transformed to a **macro** special form and subsequently expanded again.

(6) When x is an application of a derived special form, then the function M to which **(car** x**)** is bound in **Macros** will be applied to the arguments in **(quote (cdr** x**))**, giving x' = **(apply** M **(quote (cdr** x**)))**. The expanded form x' will then be expanded again.

Note that the application of M using **apply** will destructure the derived special form, so that individual parts of it will be bound to individual variables of the transformer function.

(7) All other forms x will be expanded by mapping **expand()** over their elements.

To figure out whether or not x is a derived special form, its head, **(car** x**)**, is looked up in **Macros**. x is a derived special form exactly if **(car** x**)** is bound to a transformer in that environment.

```
cell expand(cell x, int r) {
    cell    n, m;

    if (Mxlev < 0) error("interrupted", UNDEF);
    if (Mxlev > MXMAX)
        error(
          "too many levels of macro expansion",
          UNDEF);
    if (atomp(x)) {
        return x;
    }
    if (car(x) == S_quote) {
        return x;
    }
    Mxlev++;
    if (car(x) == S_lambda) {
        protect(x);
        n = mapexp(cddr(x), r);
        n = cons(cadr(x), n);
        n = cons(car(x), n);
        unprot(1);
        Mxlev--;
        return n;
    }
    if (car(x) == S_defun || car(x) == S_defmac)
    {
        protect(x);
        x = expanddef(x);
        car(Protected) = x;
        x = expand(x, r);
        unprot(1);
        Mxlev--;
        return x;
    }
```

```
    if (symbolp(car(x)) &&
         (m = assq(car(x), Macros)) != NIL)
    {
        protect(x);
        n = cons(cdr(x), NIL);
        n = cons(S_quote, n);
        n = cons(n, NIL);
        n = cons(cdr(m), n);
        n = cons(S_apply, n);
        x = eval(n, 1);
        car(Protected) = x;
        if (r) x = expand(x, r);
        unprot(1);
        Mxlev--;
        return x;
    }
    x = mapexp(x, r);
    Mxlev--;
    return x;
}
```

16. Built-In Functions

This chapter will describe functions that are used to implement *inlined primitive functions*. The simpler functions, like **cons** or **car**, will be built directly into the interpreter loop of the abstract machine, but for the more complex functions, like **bitop**, **vconc**, or **dump-image**, this approach would not be practicable. In some cases, the decision whether to inline the code in the abstract machine or factor it out to this chapter was rather arbitrary. For instance, the numerical comparison predicates could as well be inlined and others, like **abs** or **strnum**, could as well be factored out. These decisions probably do not have any great influence on performance or comprehensibility either way.

All of the functions discussed in this chapter receive their arguments as C function arguments and return their results to the caller as a function return value. For most of them, no knowledge of the underlying abstract machine is needed. The only exceptions are the **bitop** function and the fixnum and char comparison predicates (**=**, **<**, **c=**, **c<**, etc), which leave intermediate results on the runtime stack of the abstract machine.

16.1 Arithmetic Functions

The following functions implement *fixnum arithmetic*. In LISP9, integer arithmetic is performed on fixnums exclusively, there are no unbounded integers (“bignums”). However, fixnums cannot overflow; whenever a fixnum overflow would occur, the current computation is aborted and an error is signalled.

The **fixover()** function signals a *fixnum overflow* error. The *who* argument names the caller of the function, and x and y are the arguments that caused the overflow.

```
void fixover(char *who, cell x, cell y) {
	char	b[100];

	sprintf(b, "%s: fixnum overflow", who);
	error(b, cons(x, cons(y, NIL)));
```

```
}
```

The **add()** function adds the fixnums x and y and returns their sum as another fixnum. When $x + y$ would cause an integer overflow, it will signal an error.

```
cell add(cell x, cell y) {
    if (!fixp(x)) expect("+", "fixnum", x);
    if (!fixp(y)) expect("+", "fixnum", y);
    if (add_ovfl(fixval(x), fixval(y)))
        fixover("+", x, y)
    return mkfix(fixval(x) + fixval(y));
}
```

The **xsub()** ("exchange and subtract") function subtracts the fixnum x from y and returns their difference as another fixnum. When $y - x$ would cause an integer overflow, it will signal an error.

The function computes $y - x$ (not $x - y$), because the compiler will emit its arguments in reverse order. See page 219.

```
cell xsub(cell x, cell y) {
    if (!fixp(x)) expect("-", "fixnum", x);
    if (!fixp(y)) expect("-", "fixnum", y);
    if (sub_ovfl(fixval(y), fixval(x)))
        fixover("+", y, x);
    return mkfix(fixval(y) - fixval(x));
}
```

The **mul()** function multiplies the fixnums x and y and returns their product as another fixnum. When $x \times y$ would cause an integer overflow, it will signal an error.

Unfortunately, testing for integer overflow during multiplication is not trivial in C, and even the below solution sacrifices some correctness for performance. It errs on the "right" side, though, and always report an overflow when one occurs.

In machine language, the program might just multiply $x \times y$, giving r and then divide r/y and make sure the result is x. In C, however, an overflow during $x \times y$ may run nethack, launch an ICBM, or set your monitor on fire. Even worse, it may return *any* result, or even each time a different one. It is *undefined*.

So the **mul()** function goes to some length (but not the full distance) to figure out that an overflow will happen *before* it happens.

First, it captures some special cases, like $1 \times x$ or $0 \times y$ and returns their result immediately. No overflow can ever happen in these cases.

On two's complement machines, an overflow will happen when multiplying the least integer value (**INT_MIN**) by -1, because it does not have any positive counterpart, i.e. $-1 \times INT_MIN$ is undefined. This means that **abs(INT_MIN)** is also undefined.

In the next statement, **mul()** is taking a shortcut instead of sorting out the cases properly. Because the last test involves **abs**, it just assumes that any multiplication involving **INT_MIN** will overflow at this point. Note that this also means that all computations resulting in **INT_MIN** (except for $1 \times INT_MIN$ and $INT_MIN \times 1$) will signal an overflow error, even if they would in fact give a valid result. E.g. $2 \times (INT_MIN/2)$ will overflow!

The last test makes sure that $|x| \times |y| \le INT_MAX$. Only when all tests are passed, the function will actually perform the multiplication and return a result.

Considering the cases with a result of INT_MIN is left as an exercise to the enthusiastic reader.

```
cell mul(cell x, cell y) {
    int     a, b;

    if (!fixp(x)) expect("*", "fixnum", x);
    if (!fixp(y)) expect("*", "fixnum", y);
    a = fixval(x);
    b = fixval(y);
    /*
```

```
     * Overflow of a*b is undefined, sooo
     */
    /* Shortcuts, also protect later division */
    if (0 == a || 0 == b) return Zero;
    if (1 == a) return y;
    if (1 == b) return x;
    /* abs(INT_MIN) is undefined using two's
       complement, so */
    if (INT_MIN == a || INT_MIN == b)
        fixover("*", x, y);
    /* Catch the rest */
    /* Bug: result may not be INT_MIN */
    if (abs(a) > INT_MAX / abs(b))
        fixover("*", x, y);
    return mkfix(a * b);
}
```

The **intdiv()** function divides the fixnum x by y and returns their truncated quotient as another fixnum. When $y = 0$, it signals an error. The **intrem()** function is similar, but returns the remainder of the truncated integer division $trunc(x/y)$, i.e. it returns $x - y \times trunc(x/y)$. No integer overflow can occur in these operations.

```
cell intdiv(cell x, cell y) {
    if (!fixp(x)) expect("div", "fixnum", x);
    if (!fixp(y)) expect("div", "fixnum", y);
    if (0 == fixval(y))
        error("div: divide by zero", UNDEF);
    return mkfix(fixval(x) / fixval(y));
}

cell intrem(cell x, cell y) {
    if (!fixp(x)) expect("rem", "fixnum", x);
    if (!fixp(y)) expect("rem", "fixnum", y);
    if (0 == fixval(y))
        error("rem: divide by zero", UNDEF);
    return mkfix(fixval(x) % fixval(y));
```

```
}
```

The following functions implement the numeric comparison predicates **<**, **<=**, **=**, **>=**, and **>**. As discussed on page 220, they put their intermediate result ("true") on the runtime stack and change it to **nil** when the condition imposed by their predicate cannot be satisfied. For instance, the "equal" predicate **=** would change $\mathbf{S}[Sp-1]$ to **nil**, if $x \neq y$. It changes $\mathbf{S}[Sp-1]$ instead of $\mathbf{S}[Sp]$, because the latter is occupied by one of its arguments while the predicate evaluates.

All of these predicates receive their arguments in reverse order so, for instance, the **less()** function computes $y < x$ rather than $x < y$.

The **stackset()** macro sets the nth slot of the stack to v. This is a forward declaration.

```
#define stackset(n,v)    (vector(Rts)[n] = (v))

void grtr(cell x, cell y) {
    if (!fixp(x)) expect(">", "fixnum", x);
    if (!fixp(y)) expect(">", "fixnum", y);
    if (fixval(y) <= fixval(x))
        stackset(Sp-1, NIL);
}

void gteq(cell x, cell y) {
    if (!fixp(x)) expect(">=", "fixnum", x);
    if (!fixp(y)) expect(">=", "fixnum", y);
    if (fixval(y) < fixval(x))
        stackset(Sp-1, NIL);
}

void less(cell x, cell y) {
    if (!fixp(x)) expect("<", "fixnum", x);
    if (!fixp(y)) expect("<", "fixnum", y);
    if (fixval(y) >= fixval(x))
        stackset(Sp-1, NIL);
}
```

```
void lteq(cell x, cell y) {
    if (!fixp(x)) expect("<=", "fixnum", x);
    if (!fixp(y)) expect("<=", "fixnum", y);
    if (fixval(y) > fixval(x))
        stackset(Sp-1, NIL);
}

void equal(cell x, cell y) {
    if (!fixp(x)) expect("=", "fixnum", x);
    if (!fixp(y)) expect("=", "fixnum", y);
    if (fixval(y) != fixval(x))
        stackset(Sp-1, NIL);
}
```

The **bitop()** function performs the bit operation specified in the argument o on the fixnums y and x. A summary of the operations can be found in the appendix (pg 457) or they can be looked up in the code of the function.

The operation carried out by the function is $y\ o\ x$ because the operands are supplied in reverse order by the compiler. Note: o does not indicate function composition here, but the **o**peration to be performed by **bitop**.

```
cell bitop(cell x, cell y, cell o) {
    uint    op, a, b;
    int     i;

    if (!fixp(o)) expect("bitop", "fixnum", o);
    if (!fixp(x)) expect("bitop", "fixnum", x);
    if (!fixp(y)) expect("bitop", "fixnum", y);
    op = fixval(o);
    b = fixval(x);
    a = i = fixval(y);
    switch (op) {
    case  0: a =  0;        break;
    case  1: a =   a &  b;  break;
```

```
    case  2: a =   a & ~b;  break;
    case  3: /* a =   a; */ break;
    case  4: a =  ~a &  b;  break;
    case  5: a =        b;  break;
    case  6: a =   a ^  b;  break;
    case  7: a =   a |  b;  break;
    case  8: a = ~(a |  b); break;
    case  9: a = ~(a ^  b); break;
    case 10: a =       ~b;  break;
    case 11: a =   a | ~b;  break;
    case 12: a =  ~a;       break;
    case 13: a =  ~a |  b;  break;
    case 14: a = ~(a &  b); break;
    case 15: a = ~0;        break;
    case 16: a = a  <<  b;  break;
    case 17: a = i  >>  b;  break;
    case 18: a = a  >>  b;  break;
    default: error("bitop: invalid opcode", o);
         break;
    }
    return mkfix(a);
}
```

16.2 Character Functions

The functions described here operate on *char* objects. The following functions implement the predicates **c<**, **c<=**, **c=**, **c>=**, and **c>**. They work in the same way as the corresponding fixnum predicates, which were discussed previously.

```
void cless(cell x, cell y) {
    if (!charp(x)) expect("c<", "char", x);
    if (!charp(y)) expect("c<", "char", y);
    if (charval(y) >= charval(x))
         stackset(Sp-1, NIL);
}

void clteq(cell x, cell y) {
```

```
    if (!charp(x)) expect("c<=", "char", x);
    if (!charp(y)) expect("c<=", "char", y);
    if (charval(y) > charval(x))
        stackset(Sp-1, NIL);
}

void cequal(cell x, cell y) {
    if (!charp(x)) expect("c=", "char", x);
    if (!charp(y)) expect("c=", "char", y);
    if (charval(y) != charval(x))
        stackset(Sp-1, NIL);
}

void cgrtr(cell x, cell y) {
    if (!charp(x)) expect("c>", "char", x);
    if (!charp(y)) expect("c>", "char", y);
    if (charval(y) <= charval(x))
        stackset(Sp-1, NIL);
}

void cgteq(cell x, cell y) {
    if (!charp(x)) expect("c>=", "char", x);
    if (!charp(y)) expect("c>=", "char", y);
    if (charval(y) < charval(x))
        stackset(Sp-1, NIL);
}
```

The **whitespc()** macro returns “truth”, when its argument c is a *white space character* in the sense of the **whitec** predicate of LISP9.

```
#define whitespc(c) \
    (' '  == (c) || \
     '\t' == (c) || \
     '\n' == (c) || \
     '\r' == (c) || \
     '\f' == (c))
```

16.3 String Functions

The following functions process *strings*. The **scomp()** function compares two string objects and returns the difference between the first two non-matching characters in the strings x and y. When the strings have different lengths, it compares them up to the length of the shorter string plus one. The final character of the shorter string is assumed to be NUL. When the strings are equal, **scomp()** returns 0.

For example,

x	y	**returns**
`"foo"`	`"fob"`	> 0
`"fob"`	`"foo"`	< 0
`"foo"`	`"foo"`	0
`"fob"`	`"f"`	> 0

scomp() is similar to **strcmp()**, but compares string objects rather than C strings.

```
int scomp(cell x, cell y) {
    int     kx, ky;

    kx = stringlen(x);
    ky = stringlen(y);
    if (kx == ky)
        return memcmp(string(x), string(y), kx);
    return memcmp(string(x), string(y),
                  1+(kx<ky? kx: ky));
}
```

The **memcmp_ci()** function compares the first k characters in the memory regions a and b and returns the difference between the first two characters that are not equal. When all k character pairs match, it returns 0. This function is like **memcmp()**, but ignores the case of any letters contained in the regions.

```
int memcmp_ci(char *a, char *b, int k) {
```

```
    int     i, d;

    for (i=0; i<k; i++) {
        d = tolower(a[i]) - tolower(b[i]);
        if (d) return d;
    }
    return 0;
}
```

The **scomp_ci()** function is like **scomp()**, but ignores the case of any letters in its arguments. E.g., x = **"foo"** and y = **"FOO"** will give 0.

```
int scomp_ci(cell x, cell y) {
    int     kx, ky;

    kx = stringlen(x);
    ky = stringlen(y);
    if (kx == ky)
        return memcmp_ci((char *) string(x),
                         (char *) string(y),
                         kx);
    return memcmp_ci((char *) string(x),
                     (char *) string(y),
                     1+(kx<ky? kx: ky));
}
```

The following functions implement the **s<**, **s<=**, **s=**, **s>=**, and **s>** predicates, which compare strings using **scomp()**. They do not work like the numeric and char comparison predicates, because these functions are not variadic. Hence the following functions receive their argument in the proper order and return their results directly (and not via the runtime stack).

```
cell sless(cell x, cell y) {
    if (!string(x)) expect("s<", "string", x);
    if (!string(y)) expect("s<", "string", y);
```

```
    return scomp(x, y) < 0? TRUE: NIL;
}

cell slteq(cell x, cell y) {
    if (!string(x)) expect("s<=", "string", x);
    if (!string(y)) expect("s<=", "string", y);
    return scomp(x, y) <= 0? TRUE: NIL;
}

cell sequal(cell x, cell y) {
    if (!string(x)) expect("s=", "string", x);
    if (!string(y)) expect("s=", "string", y);
    if (stringlen(x) != stringlen(y)) return NIL;
    return scomp(x, y) == 0? TRUE: NIL;
}

cell sgrtr(cell x, cell y) {
    if (!string(x)) expect("s>", "string", x);
    if (!string(y)) expect("s>", "string", y);
    return scomp(x, y) > 0? TRUE: NIL;
}

cell sgteq(cell x, cell y) {
    if (!string(x)) expect("s>=", "string", x);
    if (!string(y)) expect("s>=", "string", y);
    return scomp(x, y) >= 0? TRUE: NIL;
}
```

The following functions implement the **si<**, **si<=**, **si=**, **si>=**, and **si>** predicates, which compare strings using **scomp_ci()**. They are like the above function, but ignore the case of any letters contained in their arguments.

```
cell siless(cell x, cell y) {
    if (!string(x)) expect("si<", "string", x);
    if (!string(y)) expect("si<", "string", y);
    return scomp_ci(x, y) < 0? TRUE: NIL;
}
```

```
cell silteq(cell x, cell y) {
    if (!string(x)) expect("si<=", "string", x);
    if (!string(y)) expect("si<=", "string", y);
    return scomp_ci(x, y) <= 0? TRUE: NIL;
}

cell siequal(cell x, cell y) {
    if (!string(x)) expect("si=", "string", x);
    if (!string(y)) expect("si=", "string", y);
    if (stringlen(x) != stringlen(y)) return NIL;
    return scomp_ci(x, y) == 0? TRUE: NIL;
}

cell sigrtr(cell x, cell y) {
    if (!string(x)) expect("si>", "string", x);
    if (!string(y)) expect("si>", "string", y);
    return scomp_ci(x, y) > 0? TRUE: NIL;
}

cell sigteq(cell x, cell y) {
    if (!string(x)) expect("si>=", "string", x);
    if (!string(y)) expect("si>=", "string", y);
    return scomp_ci(x, y) >= 0? TRUE: NIL;
}
```

The **b_mkstr()** ("built-in mkstr") function implements the LISP9 function **mkstr**. It has a **b_** *prefix*, because the name **mkstr()** is already used for a more low-level interface to string creation (see page 69).

b_mkstr() returns a string of the length x filled with character a. There is no single-argument version of **mkstr** internally. The default for a is generated by the compiler when the second argument is omitted.

```
cell b_mkstr(cell x, cell a) {
    cell    n;
```

```
    int     i, c, k;
    byte    *s;

    if (!fixp(x)) expect("mkstr", "fixnum", x);
    if (!charp(a)) expect("mkstr", "char", a);
    c = charval(a);
    k = fixval(x);
    n = mkstr(NULL, k);
    s = string(n);
    for (i=0; i<k; i++) s[i] = c;
    return n;
}
```

The **sconc()** function expects a list of strings *x* and returns a fresh string containing the concatenation of all strings in *x* in the same order. It implements the variadic **sconc** function.

```
cell sconc(cell x) {
    cell    p, n;
    int     k, m;
    byte    *s;

    k = 0;
    for (p = x; p != NIL; p = cdr(p)) {
        if (!stringp(car(p)))
            expect("sconc", "string", car(p));
        k += stringlen(car(p))-1;
    }
    n = mkstr(NULL, k);
    s = string(n);
    k = 0;
    for (p = x; p != NIL; p = cdr(p)) {
        m = stringlen(car(p));
        memcpy(&s[k], string(car(p)), m);
        k += m-1;
    }
    return n;
}
```

The **sref()** function extracts the nth character from the string x, where n must be a fixnum. It returns the character as a char object or signals an error when n is not a valid index into x. This function implements **sref**.

```
cell sref(cell s, cell n) {
    int     i;

    if (!stringp(s)) expect("sref", "string", s);
    if (!fixp(n)) expect("sref", "fixnum", n);
    i = fixval(n);
    if (i < 0 || i >= stringlen(s)-1)
        error("sref: index out of range", n);
    return mkchar(string(s)[i]);
}
```

The **sset()** function changes the nth character of the string s to r, where n must be a fixnum and r must be a char. The string s will be modified! An error will be signalled when n is not a valid index into s or s is immutable. This function implements **sset**.

```
void sset(cell s, cell n, cell r) {
    int     i;

    if (!stringp(s)) expect("sset", "string", s);
    if (constp(s)) error("sset: immutable", s);
    if (!fixp(n)) expect("sset", "fixnum", n);
    if (!charp(r)) expect("sset", "char", r);
    i = fixval(n);
    if (i < 0 || i >= stringlen(s)-1)
        error("sset: index out of range", n);
    string(s)[i] = charval(r);
}
```

The **substr()** function creates a fresh string containing the

characters from position n_0 up to, but not including, position n_1 of the string s. It will signal an error, if the following assertion does not hold:

$0 \le n_0 \le n_1 \le |s|$

where $|s|$ is the length of s. This function implements the LISP9 function **substr**.

```
cell substr(cell s, cell n0, cell n1) {
    int     k, k0, k1, i, j;
    cell    n;
    byte    *s0, *s1;

    if (!stringp(s))
        expect("substr", "string", s);
    if (!fixp(n0))
        expect("substr", "fixnum", n0);
    if (!fixp(n1))
        expect("substr", "fixnum", n1);
    k0 = fixval(n0);
    k1 = fixval(n1);
    if (k0 < 0 || k1 < 0 || k0 > k1 ||
        k1 >= stringlen(s)
    )
        error("substr: invalid range",
              cons(n0, cons(n1, NIL)));
    k = k1-k0;
    n = mkstr(NULL, k);
    j = 0;
    s0 = string(s);
    s1 = string(n);
    for (i=k0; i<k1; i++) {
        s1[j] = s0[i];
        j++;
    }
    s1[j] = 0;
    return n;
}
```

The **sfill()** function fills all positions of the string x with the character a. The string x will be modified! An error will be signalled, if the string is immutable. This function implements **sfill**.

```
void sfill(cell x, cell a) {
    int     c, i, k;
    byte    *s;

    if (!stringp(x))
        expect("sfill", "string", x);
    if (constp(x))
        error("sfill: immutable", x);
    if (!charp(a))
        expect("sfill", "char", a);
    c = charval(a);
    k = stringlen(x)-1;
    s = string(x);
    for (i=0; i<k; i++) s[i] = c;
}
```

16.4 Vector Functions

The following functions process *vectors*. They are mostly identical to the functions processing strings, which should not be a surprise, because strings are essentially vectors of bytes.

The **b_mkvec()** function returns a fresh vector of the length x filled with the object a. It implements **mkvec**.

```
cell b_mkvec(cell x, cell a) {
    cell    n;
    int     i, k;
    cell    *v;

    if (!fixp(x)) expect("mkvec", "fixnum", x);
    k = fixval(x);
    n = mkvec(k);
```

```
    v = vector(n);
    for (i=0; i<k; i++) v[i] = a;
    return n;
}
```

The **vconc()** function expects a list of vectors x and returns a fresh vector containing the concatenation of the vectors in x. It implements **vconc**.

```
cell vconc(cell x) {
    cell    p, n, *v;
    int     k, m;

    k = 0;
    for (p = x; p != NIL; p = cdr(p)) {
        if (!vectorp(car(p)))
            expect("vconc", "vector", car(p));
        k += veclen(car(p));
    }
    n = mkvec(k);
    v = vector(n);
    k = 0;
    for (p = x; p != NIL; p = cdr(p)) {
        m = veclen(car(p));
        memcpy(&v[k], vector(car(p)),
                m*sizeof(cell));
        k += m;
    }
    return n;
}
```

The **vref** function extracts the nth element of the vector x and returns it. $0 \leq n < |x|$ must hold. This function implements **vref**.

```
cell vref(cell x, cell n) {
    int     i;
```

```
    if (!vectorp(x)) expect("vref", "vector", x);
    if (!fixp(n)) expect("vref", "fixnum", n);
    i = fixval(n);
    if (i < 0 || i >= veclen(x))
        error("vref: index out of range", n);
    return vector(x)[i];
}
```

The **vfill()** function fills every slot of the vector x with the object a. The vector must be mutable. This function implements **vfill**.

```
void vfill(cell x, cell a) {
    int     i, k;
    cell    *v;

    if (!vectorp(x))
        expect("vfill", "vector", x);
    if (constp(x))
        error("vfill: immutable", x);
    k = veclen(x);
    v = vector(x);
    for (i=0; i<k; i++) v[i] = a;
}
```

The **vset()** function changes the nth slot of the vector v to r. v must be mutable and $0 \le n < |v|$ must hold. This function implements **vset**.

```
void vset(cell v, cell n, cell r) {
    int     i;

    if (!vectorp(v)) expect("vset", "vector", v);
    if (constp(v)) error("vset: immutable", v);
    if (!fixp(n)) expect("vset", "fixnum", n);
    i = fixval(n);
    if (i < 0 || i >= veclen(v))
        error("vset: index out of range", n);
```

```
    vector(v)[i] = r;
}
```

The **subvec()** function creates a fresh vector containing the objects from slot n_0 up to, but not including, slot n_1 of the vector v. The assertion $0 \le n_0 \le n_1 \le |v|$ must hold. This function implements **subvec**.

```
cell subvec(cell v, cell n0, cell n1) {
    int     k, k0, k1, i, j;
    cell    n;
    cell    *v0, *v1;

    if (!vectorp(v))
        expect("subvec", "vector", v);
    if (!fixp(n0))
        expect("subvec", "fixnum", n0);
    if (!fixp(n1))
        expect("subvec", "fixnum", n1);
    k0 = fixval(n0);
    k1 = fixval(n1);
    if (k0 < 0 || k1 < 0 || k0 > k1 ||
        k1 > veclen(v)
    )
        error("subvec: invalid range",
              cons(n0, cons(n1, NIL)));
    k = k1-k0;
    n = mkvec(k);
    j = 0;
    v0 = vector(v);
    v1 = vector(n);
    for (i=k0; i<k1; i++) {
        v1[j] = v0[i];
        j++;
    }
    return n;
```

16.5 Input/Output Functions

The **existsp()** predicate returns **T**, if the file specified in *s* exists. It implements the LISP9 function **existsp**.

```
cell existsp(char *s) {
    FILE    *f;

    f = fopen(s, "r");
    if (f != NULL) fclose(f);
    return NULL == f? NIL: TRUE;
}
```

The **openfile()** function opens the file specified in the string object *x* in the mode specified in *mode*, where a mode of 0 means input, 1 means output, and 2 equals 1 but output will be *appended* to the given file.

This function is used to implement the LISP9 functions **open-infile** and **open-outfile**.

```
cell openfile(cell x, int mode) {
    int     p;

    switch (mode) {
    case 0:
        p = open_inport((char *) string(x));
        break;
    case 1:
        p = open_outport((char *) string(x), 0);
        break;
    case 2:
        p = open_outport((char *) string(x), 1);
        break;
    }
    if (p < 0) {
        if (0 == mode)
```

```
            error("open-infile: cannot open", x);
        else
            error("open-outfile: cannot open", x);
    }
    return mkport(p, 0 == mode? T_INPORT:
                                 T_OUTPORT);
}
```

The **b_readc()** function implements **readc** and **peekc**. It reads a single character from the port *p* and returns it as a char object. When no input is available on *p*, it returns the EOF marker. When the *rej* ("reject") argument is non-zero, it will reject a character after reading it (i.e., it will implement **peekc**).

```
cell b_readc(cell p, int rej) {
    int     pp, c;

    pp = Inport;
    if (p != pp) set_inport(p);
    c = readc();
    if (rej) rejectc(c);
    if (p != pp) set_inport(pp);
    if (EOF == c) return EOFMARK;
    return mkchar(c);
}
```

The **b_read()** function is a tiny wrapper around the LISP reader, **read**. When its argument *ps* is a port, it will read from that port, and if *ps* is a string, it will parse that string. It will return the object it has read or the EOF marker.

```
cell b_read(cell ps) {
    int     pp;
    cell    n;

    if (stringp(ps)) {
        Instr = (char *) string(ps);
```

```
        Rejected = -1;
        n = xread();
        Instr = NULL;
        if (Readerr)
            return mkstr(Readerr,
                         strlen(Readerr));
        return cons(n, NIL);
    }
    ps = portno(ps);
    pp = Inport;
    if (ps != pp) set_inport(ps);
    n = xread();
    if (ps != pp) set_inport(pp);
    return n;
}
```

The **b_prin()** function is a tiny wrapper around the LISP9 functions **prin** and **princ**. It prints its object x to the port p. When its sl ("slashify") argument is true, it will print the external representation of x and otherwise it will pretty-print x.

```
void b_prin(cell x, int p, int sl) {
    int     pp;

    pp = Outport;
    if (p != pp) set_outport(p);
    prex(sl, x, 0);
    if (p != pp) set_outport(pp);
}
```

The **format()** function is a tiny wrapper around **format**. It writes the external representation of x to a fresh string and returns it.

```
cell format(cell x) {
    cell    n;

    Outstr = mkstr(NULL, 1000);
```

```
    Outmax = 1000;
    Outptr = 0;
    prex(1, x, 0);
    n = mkstr(NULL, Outptr);
    memcpy(string(n), string(Outstr), Outptr+1);
    Outstr = NIL;
    return n;
}
```

The **writec()** function writes the character c to port p. It implements **writec**.

```
void b_writec(int c, cell p) {
    int     pp;

    pp = Outport;
    if (p != pp) set_outport(p);
    writec(c);
    if (p != pp) set_outport(pp);
}
```

16.6 List Functions

The **lconc()** function is like **sconc** or **vconc**, but concatenates lists. Its argument x must be a list of proper lists (except for the last element, which may be a dotted list or an atom). **lconc()** creates a fresh list. It implements the **conc** function.

```
cell lconc(cell x) {
    cell    p, q, n, m;
    int     k;

    if (NIL == cdr(x)) return car(x);
    protect(n = cons(NIL, NIL));
    k = 0;
    for (p = x; cdr(p) != NIL; p = cdr(p)) {
        if (NIL == car(p)) continue;
```

```
        for (q = car(p); q != NIL; q = cdr(q)) {
            if (!pairp(q))
                expect("conc", "list", car(p));
            if (k != 0) {
                m = cons(NIL, NIL);
                cdr(n) = m;
                n = cdr(n);
            }
            car(n) = car(q);
            k++;
        }
    }
    m = unprot(1);
    if (0 == k) return car(p);
    cdr(n) = car(p);
    return m;
}
```

The **nlconc()** function concatenates the lists in the list x *destructively* by setting the last cdr field of the first list to the second list, the last cdr field of the second list to the third list, and so on.

All elements of x (except for the last one) must be proper lists and must be mutable. **nlconc()** implements **nconc**.

```
cell nlconc(cell x) {
    cell    p, q;

    while (pairp(cdr(x)) && NIL == car(x))
        x = cdr(x);
    if (NIL == cdr(x)) return car(x);
    for (p = x; cdr(p) != NIL; p = cdr(p)) {
        if (NIL == car(p)) continue;
        if (constp(car(p)))
            error("nconc: immutable", car(p));
        for (q = car(p);
             cdr(q) != NIL;
```

```
                q = cdr(q))
        {
            if (!pairp(q))
                expect("nconc", "list", car(p));
        }
        while (pairp(cdr(p)) && NIL == cadr(p))
            p = cdr(p);
        if (NIL == cdr(p)) break;
        cdr(q) = cadr(p);
    }
    return car(x);
}
```

16.7 Type Conversion Functions

The **b_symbol()** function converts the string x to an interned *symbol*. It implements the **symbol** function.

Note that the code

```
mkstr(symname(x), symlen(x))
```

would not work, because **symname(x)** creates a reference to the vector pool and the subsequent **mkstr()** application can trigger a garbage collection, which in turn may move the symbol name of x to a different location.

```
cell b_symbol(cell x) {
    cell    y, n, k;

    y = findsym((char *) string(x));
    if (y != NIL) return y;
    /*
     * Cannot pass content to mksym(), because
     * string(x) may move during GC.
     */
    k = stringlen(x);
    n = mksym("", k-1);
    memcpy(symname(n), string(x), k);
    return intern(n);
```

```
}
```

The **symname()** function returns a string containing the name of the symbol x. It implements the **symname** function. The same caveat as above applies.

```
cell b_symname(cell x) {
    cell    n, k;

    /*
     * Cannot pass name to mkstr(), because
     * symname(x) may move during GC.
     */
    k = symlen(x);
    n = mkstr(NULL, k-1);
    memcpy(string(n), symname(x), k);
    return n;
}
```

The **liststr()** function returns a fresh string containing the characters passed to it in the list x. The list must be a proper list and contain char objects exclusively. The function implements **liststr**.

```
cell liststr(cell x) {
    cell    n, v;
    int     k;
    byte    *s;

    k = 0;
    for (n = x; n != NIL; n = cdr(n))
        k++;
    v = mkstr(NULL, k);
    s = string(v);
    for (n = x; n != NIL; n = cdr(n)) {
        if (atomp(n))
            error("liststr: dotted list", x);
```

```
        if (!charp(car(n)))
            expect("liststr", "char", car(n));
        *s = charval(car(n));
        s++;
    }
    return v;
}
```

The **listvec()** function returns a fresh vector containing the objects passed to it in the (proper!) list x. It implements **listvec**.

When the *veclit* argument is non-zero, the function is being called in order to parse a *vector literal* and otherwise is implements the **listvec** function. When parsing a vector literal, the resulting object will be immutable.

```
cell listvec(cell x, int veclit) {
    cell    n, v;
    int     k;
    cell    *p;
    char    *msg;

    msg = veclit?
          "vector literal contains a dot":
          "listvec: dotted list";
    k = 0;
    for (n = x; n != NIL; n = cdr(n))
        k++;
    if (0 == k) return Nullvec;
    v = mkvec(k);
    if (veclit) tag(v) |= CONST_TAG;
    p = vector(v);
    for (n = x; n != NIL; n = cdr(n)) {
        if (atomp(n)) error(msg, x);
        *p = car(n);
        p++;
    }
    return v;
```

```
}
```

The **strlist()** function returns a fresh list containing the characters of the string x as char objects. It implements **strlist**.

```
cell strlist(cell x) {
    cell    a, new;
    int     k, i;

    k = stringlen(x)-1;
    if (0 == k) return NIL;
    protect(a = cons(NIL, NIL));
    for (i=0; i<k; i++) {
        new = mkchar(string(x)[i]);
        car(a) = new;
        if (i < k-1) {
            new = cons(NIL, NIL);
            cdr(a) = new;
            a = cdr(a);
        }
    }
    return unprot(1);
}
```

The **veclist()** function returns a fresh list containing the objects of the vector x. It implements **veclist**.

```
cell veclist(cell x) {
    cell    a, new;
    int     k, i;

    k = veclen(x);
    if (0 == k) return NIL;
    protect(a = cons(NIL, NIL));
    for (i=0; i<k; i++) {
        car(a) = vector(x)[i];
        if (i < k-1) {
```

```
            new = cons(NIL, NIL);
            cdr(a) = new;
            a = cdr(a);
        }
    }
    return unprot(1);
}
```

The **numstr()** function returns a string containing the canonical external representation of the *fixnum* x to the base of r. It implements the **numstr** function. The assertion $2 \le r \le 36$ must hold.

```
cell numstr(cell x, int r) {
    char    *p;

    if (r < 2 || r > 36)
        error("numstr: bad radix", mkfix(r));
    p = ntoa(fixval(x), r);
    return mkstr(p, strlen(p));
}
```

The **strnum()** function attempts to parse the string s as a fixnum. When the characters in x indeed form the external representation of a fixnum to the base of r, the value of the number is returned as a fixnum object. Otherwise, the function returns **nil**. The assertion $2 \le r \le 36$ must hold. **strnum()** implements LISP9's **strnum** function.

```
cell strnum(char *s, int r) {
    if (r < 2 || r > 36)
        error("strnum: bad radix", mkfix(r));
    return scanfix(s, r, 0);
}
```

16.8 Program Loading

The **loadfile()** function loads LISP expressions from the file specified in the argument *s*. *Loading* an expression means to read and evaluate it. No values will print while loading. The function implements the following procedure:

- The input file *s* is opened; if it cannot be opened, an error will be signalled.
- The input port linked to the file is locked, so it cannot be closed by the garbage collector while loading.
- The current input port is memorized and changed (*rdport*).
- The current input file is registered (*Files*).
- The line counter is memorized (*oline*) and reset.
- Expressions are read from the input file and then evaluated with the current input port connected to *rdport*. This means that loading **(read)** will read the input port from which loading was initiated and *not* the file being loaded.
- When the EOF of the input file is reached, the input file is unregistered, the line counter reset to its previous state, the input file is closed (thereby unlocking it), and the original current input port is restored.

The **begin_rec()** and **end_rec()** functions, which are forward-declared here, make the **eval()** function reentrant by saving and restoring its global state. **loadfile()** saves the state *once* before loading a complete file and restores it once after loading. Therefore it can call **eval()** with the recursion flag cleared, which reduces consing at load time.

The **loadfile()** function is used to implement the following:

- the LISP function **load**
- loading libraries from the command line
- batch evaluation
- compilation of the initial LISP image

```
void    begin_rec(void);
void    end_rec(void);

void loadfile(char *s) {
    int     ldport, rdport, oline;
    cell    x;

    ldport = open_inport(s);
    if (ldport < 0)
        error("load: cannot open file",
            mkstr(s, strlen(s)));
    lock_port(ldport);
    rdport = Inport;
    oline = Line;
    Files = cons(mkstr(s, strlen(s)), Files);
    Line = 1;
    begin_rec();
    for (;;) {
        set_inport(ldport);
        x = xread();
        set_inport(rdport);
        if (EOFMARK == x) break;
        eval(x, 0);
    }
    end_rec();
    Files = cdr(Files);
    Line = oline;
    close_port(ldport);
}
```

The **load()** function loads the file specified in the string *x*. It implements **load**.

```
void load(cell x) {
    char    path[TOKLEN+1];

    if (!stringp(x))
        expect("load", "string", x);
```

```
    if (stringlen(x) > TOKLEN)
        error("load: path too long", x);
    strcpy(path, (char *) string(x));
    loadfile(path);
}
```

16.9 Heap Images

LISP has been an *image based* system from the beginning [McCarthy1960b]. An *image* is basically a copy of the memory that is used by the LISP system that is being written to mass storage. Today, running different programs one after another, or even simultaneously, is a common occasion. Back in the days when LISP was new technology, computers typically ran one job at a time, which typically involved mounting a tape and loading the program before it could be used. Once the LISP system was loaded from tape, it would also have to load and compile all the functions that a LISP program needed, which would take even more time and probably involve the installation of another tape.

So once the LISP system was completely loaded and ready to operate, an image file was written to a different tape. The entire LISP system could then be loaded from the image tape and would be operational immediately. When changes were made to the LISP system, the image tape could be overwritten, so all modifications made in one session would be available immediately in the next session.

Even today it sometimes makes sense to use image files, because they can reduce program loading times significantly, especially for large programs.

The state saved in a typical image file includes

- all interned symbols
- the global environment
- all literal objects
- all macro definitions
- all compiled code

- the free lists
- the memory pools

When loading a saved image, the system is exactly in the same state as it was when the image was saved. The process of saving an image is sometimes referred to as *image dumping*.

Multiple images for the same LISP system can exist. For instance, there may be an image that contains the on-line help system and another one that contains the disassembler or code grinder (pretty printer).

Images can be dumped at any time, but they can usually be loaded only during initialization, when the LISP system comes up.

The LISP9 design makes saving and loading image files an easy process. The abstract machine code is relocatable and references to data structures are offsets rather than pointers, so they cannot fall prey to *address space layout randomization* [Spengler2003], which loads program segments at random addresses, thereby invalidating any pointers contained in an image file.

Dumping a LISP9 image is as simple as writing the node pool, the vector pool, and some administrative data to a file. Loading an image file is the reverse process.

The **imghdr** structure is the *header* of the LISP9 image file format. It identifies and describes the image file in which it is contained. It allocates the first 32 bytes in the file.

The *magic* field of the header contains the *magic number* or *magic ID* of the file. This part is used to identify the file. When the first bytes of the file are not equal to the magic ID, the LISP system will refuse to load the file. The LISP9 magic ID is **"LISP9"**.

The "*version*" field contains the release date of the LISP system in ASCII digits using the ISO 8061 date format ("yyyymmdd").

The *cell_size* field holds the size of a **cell** as a single ASCII digit. In the current implementation, it will always be **"4"**.

The *byte_order* field contains the ASCII digits **"1"**, **"2"**, **"3"**, **"4"** in the byte ordering of the host system running the LISP process. On a little-endian system, this will be **"4321"**, on a big-endian

system **"1234"**.

Images will only load, if the *version*, *cell_size*, and *byte_order* fields of the system and the image are identical.

```
struct imghdr {
    char    magic[5];         /* "LISP9"       */
    char    version[8];       /* "yyyymmdd"    */
    char    cell_size[1];     /* size + '0'    */
    char    byte_order[4];    /* e.g. "4321"   */
    char    pad[14];
};
```

The **xfwrite()** function is like **fwrite()**, but returns a string describing a file write error in case not all bytes of its buffer could be written to storage. It returns **NULL** on success.

```
char *xfwrite(void *buf, int siz, int n, FILE *f)
{
    if (fwrite(buf, siz, n, f) != n)
        return "image file write error";
    return NULL;
}
```

The **saveimg()** function moves an old image file out of the way, so a backup copy is present when dumping a new image. Normally, LISP9 image files are named "*name*.image". This function will rename an existing image to "*name*.oimage". When the original file has no suffix, ".oimage" will be appended to the new name.

When no old image file exists or it could not be renamed for some reason, this function will fail silently. Caveat utilitor!

```
void saveimg(char *path) {
    char    b[TOKLEN+1], *p;

    if (strlen(path)+7 >= TOKLEN)
        error("image path too long", UNDEF);
```

```
    strcpy(b, path);
    p = strrchr(b, '.');
    if (NULL == p) p = b+strlen(b);
    *p = 0;
    strcat(b, ".oimage");
    remove(b);
    rename(path, b);
}
```

The **dumpimg()** function writes the current state of the LISP9 system to the image file specified in *path* after moving an existing image file out of the way. It basically writes the components listed in figure 48 sequentially to a file and makes sure that they actually have been output.

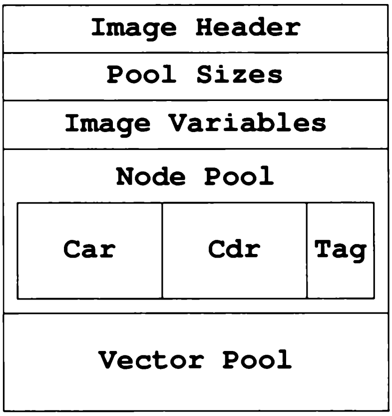


Fig. 48 – Image file format

The *pool sizes* are constant in the current implementation, but if another system is built with different pool sizes, the systems will not be able to exchange images.

The *image variables* form a set of internal (C level) variables that need to be initialized when an image is being reinstated. The **Imagevars** array, which holds pointers to these variables, is forward-declared here and defined and initialized later.

When the **dumpimg()** function fails to dump an image file, it returns a string describing the issue. In case of success, it returns **NULL**.

```
cell *Imagevars[];

char *dumpimg(char *path) {
    FILE          *f;
    cell          n, **v;
    int      i;
    struct imghdr   m;
    char          *s;

    saveimg(path);
    f = fopen(path, "wb");
    if (NULL == f)
        return "cannot create image file";
    memset(&m, '_', sizeof(m));
    strncpy(m.magic, "LISP9", sizeof(m.magic));
    strncpy(m.version, VERSION,
            sizeof(m.version));
    m.cell_size[0] = sizeof(cell)+'0';
    n = 0x31323334L;
    memcpy(m.byte_order, &n, 4);
    if ((s = xfwrite(&m, sizeof(m), 1, f))
             != NULL)
    {
        fclose(f);
        return s;
    }
    i = NNODES;
    if ((s = xfwrite(&i, sizeof(int), 1, f))
             != NULL)
    {
        fclose(f);
        return s;
    }
    i = NVCELLS;
    if ((s = xfwrite(&i, sizeof(int), 1, f))
             != NULL)
    {
```

```
        fclose(f);
        return s;
    }
    i = 0;
    v = Imagevars;
    while (v && v[i]) {
        if ((s = xfwrite(v[i], sizeof(cell),
                                1, f))
                    != NULL)
        {
            fclose(f);
            return s;
        }
        i++;
    }
    if (fwrite(Car, 1, sizeof(cell) * NNODES, f)
         != sizeof(cell) * NNODES ||
        fwrite(Cdr, 1, sizeof(cell) * NNODES, f)
         != sizeof(cell) * NNODES ||
        fwrite(Tag, 1, NNODES, f) != NNODES||
        fwrite(Vectors, 1, sizeof(cell) * NVCELLS, f)
         != sizeof(cell) * NVCELLS)
    {
        fclose(f);
        return "image dump failed";
    }
    fclose(f);
    return NULL;
}
```

The **xfread()** function is like **fread()**, but returns a string describing a file read error in case too few bytes were read from storage. It returns **NULL** on success.

```
char *xfread(void *buf, int siz, int n, FILE *f)
{
    if (fread(buf, siz, n, f) != n)
        return "image file read error";
```

```
    return NULL;
}
```

loadimg() is the inverse function of **dumpimg()**: it loads an image from mass storage and initializes the state of the LISP9 system according to the data stored in the image. It returns an error message in case of failure and **NULL** in case of success.

Basically, **loadimg()** makes sure that the data in the header match those of the LISP9 system loading the image, makes sure that the pool sizes are correct, and then loads the pools and initializes the image variables.

Note that loading an image will *overwrite* the complete state of the LISP9 system, so variables that need to be initialized at startup time will have to be initialized *after* loading an image (for instance, the ***quiet*** and ***imagefile*** variables). See chapter 19 for details.

```
char *loadimg(char *path) {
    FILE          *f;
    cell          n, **v;
    int      i;
    struct imghdr   m;
    int      image_nodes, image_vcells;
    char          *s;

    f = fopen(path, "rb");
    if (NULL == f)
        return "could not open file";
    if ((s = xfread(&m, sizeof(m), 1, f))
            != NULL)
    {
        return s;
    }
    if (memcmp(m.magic, "LISP9", sizeof(m.magic)))
    {
        fclose(f);
        return "imghdr match failed";
    }
```

```
if (memcmp(m.version, VERSION,
            sizeof(m.version)))
{
    fclose(f);
    return "wrong image version";
}
if (m.cell_size[0]-'0' != sizeof(cell)) {
    fclose(f);
    return "wrong cell size";
}
memcpy(&n, m.byte_order, sizeof(cell));
if (n != 0x31323334L) {
    fclose(f);
    return "wrong byte order";
}
memset(Tag, 0, NNODES);
if ((s = xfread(&image_nodes, sizeof(int), 1, f))
         != NULL)
    return s;
if ((s = xfread(&image_vcells, sizeof(int), 1, f))
         != NULL)
    return s;
if (image_nodes != NNODES) {
    fclose(f);
    return "wrong node pool size";
}
if (image_vcells != NVCELLS) {
    fclose(f);
    return "wrong vector pool size";
}
v = Imagevars;
i = 0;
while (v && v[i]) {
    if ((s = xfread(v[i], sizeof(cell), 1, f))
             != NULL)
    {
        return s;
    }
```

```
        i++;
    }
    if ((fread(Car, 1, sizeof(cell) * NNODES, f)
          != sizeof(cell) * NNODES ||
         fread(Cdr, 1, sizeof(cell) * NNODES, f)
          != sizeof(cell) * NNODES ||
         fread(Tag, 1, NNODES, f) != NNODES ||
         fread(Vectors, 1, sizeof(cell) * NVCELLS, f)
          != sizeof(cell) * NVCELLS ||
         fgetc(f) != EOF))
    {
        fclose(f);
        return "wrong file size";
    }
    fclose(f);
    return NULL;
}
```

The **dump_image()** function dumps an image file and signals an error, if dumping failed for any reason. Upon success, it binds the path of the image file to the variable ***imagefile***. This function implements the LISP9 function **dump-image**.

```
void dump_image(cell s) {
    char    *rc;

    rc = dumpimg((char *) string(s));
    if (rc != NULL) {
        remove((char *) string(s));
        error(rc, s);
    }
    bindset(S_imagefile, s);
}
```

16.10 Miscellaneous Functions

The **b_gc()** function triggers a garbage collection and vector pool

compaction and returns a list (n m), where n is the number of free nodes and m is the number of free vector cells. It implements the LISP function **gc**.

```
cell b_gc(void) {
    cell    n;

    gcv();
    n = cons(mkfix(NVCELLS-Freevec), NIL);
    protect(n);
    n = mkfix(length(Freelist));
    return cons(n, unprot(1));
}
```

The **gensym()** function, which implements **gensym**, generates an *uninterned* symbol of the form **G**n, where n is an integer number.

Note that the algorithm used to generate a "unique" symbol is very simple and makes no effort to actually create a symbol that is unique. Although each symbol returned by **gensym** will differ from all other symbols generated by the function in *the same session*, there is no guarantee that unique symbols will still be generated after terminating a LISP session and starting a new one. In fact, LISP9 will start generating symbols at $n = 1$ each time the system comes up. Also, **(symbol "G1")** will generate a duplicate symbol (and intern it).

The LISP9 system can get away with generating non-unique symbols, because they are only used in *local* bindings of macro bodies. Macro bodies are generated by the LISP system rather than read by **read**, so symbol identity does not matter. During macro expansion local names will be eliminated completely by the compiler, After compiling an expression like

```
((lambda (G1)
   (setq G1 (lambda nil
              (if t (prog)
                    (prog (G1)))))
   (G1))
```

```
 nil)
```

(a macro-expanded **do** expression), all references to the symbol $G1$ will be eliminated. In fact, the same symbol could be reused immediately after compiling the above. Different symbols are only generated by **gensym** to cover the case where multiple local variables with "unique" names are required in the same compilation unit.

Also note that names like $G1$ can never be captured, because the reader would translate $G1$ to $g1$. Given

```
(defmac (swap a b)
  (let ((g (gensym)))
    @(let ((,g ,a))
       (setq ,a ,b)
       (setq ,b ,g))))
```

the application

```
(swap G1 G2)
```

which might expand to

```
(let ((G1 g1))
  (setq g1 g2)
  (setq g2 G1))
```

would not capture any name, because $g1$ and $G1$ are different symbols.

```
cell gensym(void) {
    static int  id = 0;
    char        b[100];

    id++;
    sprintf(b, "G%d", id);
    return mksym(b, strlen(b));
}
```

The **untag()** function returns the "payload" of a tagged data object. In most cases, its result will be unusable or meaningless or

both. For example,

```
(untag 123)
```

will return the object **#<atom 123>**, which has no properties except for identity. The only interesting application is to apply **untag** to a function, in which case the *bytecode* of the function will be returned as a string.

The **untag()** function implements the LISP9 function **untag**.

```
cell untag(cell x) {
    if (specialp(x)) return x;
    if (tag(x) & VECTOR_TAG) return NIL;
    if (closurep(x)) return cdr(cadddr(x));
    return cdr(x);
}
```

17. The Abstract Machine

This chapter describes the implementation of the *LISP9 abstract machine* (*LAM*), as introduced in section 14.3.

The following definitions implement the *registers* of the machine as well as its *runtime stack*. To summarize the registers again:

- **Prog**, the current program fragment
- **Ip**, the instruction pointer
- **Acc**, the accumulator
- **Sp**, the stack pointer
- **Fp**, the frame pointer
- **Ep**, the environment pointer
- **E0**, the top level environment

The variable **Rts** will be bound to a vector of **CHUNKSIZE** slots that implements the runtime stack. The variable **Sz** holds the current size of the stack.

```
cell    Prog = NIL;

int     Ip = 0;

cell    Acc = NIL;

int     Sz = CHUNKSIZE;

cell    Rts = NIL;
int     Sp = -1,
        Fp = -1;

cell    E0 = NIL,
        Ep = NIL;
```

The bytecode of the abstract machine is stored in *bytecode*

containers, as shown in figure 43 on page 193. The cdr part of a bytecode container is a string that holds the instructions of a program fragment.

The **ins()** macro extracts the instruction pointed to by **Ip** from the bytecode container **Prog**. The macros **op1()** and **op2()** extract the first and second inline operand of the instruction, respectively. The operand macros will have to be adjusted when the size of the LAM operands should change.

```
#define ins()    (string(cdr(Prog))[Ip])

#define op1()   fetcharg(string(cdr(Prog)), Ip+1)
#define op2()   fetcharg(string(cdr(Prog)), Ip+3)
```

The **skip()** macro skips over an instruction of a size of n bytes by incrementing **Ip**. The **clear()** macro removes n arguments from the stack.

```
#define skip(n)      (Ip += (n))
#define clear(n)     (Sp -= (n))
```

The **box()** macro creates a fresh *location* (also called a *box*). The **boxref()** macro extracts the values from a box and **boxset()** changes the value stored in the box x to v.

```
#define box(x)       cons((x), NIL)
#define boxref(x)    car(x)
#define boxset(x,v) (car(x) = (v))
```

The **stackref()** macro extracts the value in the nth slot of the runtime stack, where the bottom of the stack has an index of 0 and the top of the stack is Sp. **stackset()** changes the value in slot n to v. Elements stored on the stack may be either boxes or values.

```
#define stackref(n)       (vector(Rts)[n])
```

```
#define stackset(n,v)    (vector(Rts)[n] = (v))
```

The **envbox()** macro references the *box* (*not* the value) of the nth box of the current environment. **argbox()** references the box in the nth argument slot, i.e. the binding of the nth argument.

```
#define envbox(n)    (vector(Ep)[n])
#define argbox(n)    (stackref(Fp-(n)))
```

The **argref()** macro extracts the value of the nth *argument* of a closure and the **arg()** macro extracts the value of the nth argument (stack *operand*) of a bytecode instruction.

```
#define argref(n)    boxref(argbox(n))
#define arg(n)       boxref(stackref(Sp-(n)))
```

The **stkalloc()** function makes sure that there is enough space on the stack to push k objects to it. In the vast majority of cases, this function will do nothing, but when $Sp + k \geq Sz$, it will allocate sufficient space to store k additional elements.

It does so by allocating a new stack, whose size will be a multiple of **CHUNKSIZE**, and copying over the "live" elements of the old stack. See figure 12 on page 56.

```
void stkalloc(int k) {
    cell    n, *vs, *vn;
    int     i;

    if (Sp + k >= Sz) {
        /* allocate multiples of CHUNKSIZE */
        if (k >= CHUNKSIZE) {
            k = Sp+k-Sz;
            k = CHUNKSIZE * (1 + k / CHUNKSIZE);
        }
        else {
```

```
            k = CHUNKSIZE;
        }
        n = mkvec(Sz + k);
        vs = vector(Rts);
        vn = vector(n);
        for (i=0; i<=Sp; i++) vn[i] = vs[i];
        Sz += k;
        Rts = n;
    }
}
```

The **push()** function pushes the value x to the runtime stack and the **pop()** function removes the top element from the stack and returns it.

An unbalanced stack (removing an element from an empty stack) should never happen. This condition would signal an internal error in the abstract machine.

```
void push(cell x) {
    Tmp = x;
    stkalloc(1);
    Tmp = NIL;
    Sp++;
    stackset(Sp, x);
}

cell pop(void) {
    if (Sp < 0) error("oops: stack underflow",
                      UNDEF);
    Sp--;
    return stackref(Sp+1);
}
```

The following functions and macros implement the *closure* data object as described on pages 174+ of the formal definition of the instruction set of the LISP9 abstract machine.

The **closure()** function corresponds to the $C(i, e, p)$ definition, which creates a fresh closure object with the given entry point (i), environment (e), and program fragment (p). However, the **closure()** function always uses the program fragment **Prog**, hence it expects only two arguments. The structure of the tagged closure data object is shown in figure 49.

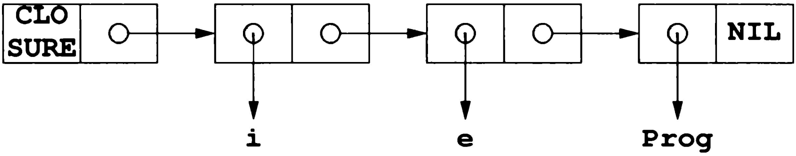


Fig. 49 – The structure of a closure object

The **closure_ip()**, **closure_env()**, and **closure_prog()** macros extract the individual components of a closure object. They resemble the definitions C_0, C_E, and C_P on pages 174+.

```
cell closure(cell i, cell e) {
    cell    c;

    c = cons(Prog, NIL);
    c = cons(e, c);
    protect(c);
    c = cons(mkfix(i), c);
    unprot(1);
    return mkatom(T_CLOSURE, c);
}

#define closure_ip(c)   cadr(c)
#define closure_env(c)  caddr(c)
#define closure_prog(c) cadddr(c)
```

The **apply()** function implements the *apply* instruction (see page 177) and, if $tail = 1$, its tail-recursive cousin, *tailapp*. It signals an error, if the object in *Acc* is not a closure.

```
int apply(int tail) {
    int     n, m, pn, pm, i;
```

```
    cell    k, e;

    if (!closurep(Acc))
        error("application of non-function",
              Acc);
    if (tail) {
        Ep = closure_env(Acc);
        Prog = closure_prog(Acc);
        m = fixval(stackref(Sp));
        n = fixval(stackref(Sp-m-4));
        pm = Sp-m;
        pn = Sp-m-n-4;
        if (n == m) {
            for (i=0; i<=m; i++)
                stackset(pn+i, stackref(pm+i));
            Fp = fixval(stackref(Sp-m-1));
            Sp -= n+2;
        }
        else {
            e = stackref(Sp-m-3);
            k = stackref(Sp-m-2);
            Fp = fixval(stackref(Sp-m-1));
            for (i=0; i<=m; i++)
                stackset(pn+i, stackref(pm+i));
            Sp -= n+2;
            stackset(Sp-1, e);
            stackset(Sp,   k);
        }
    }
    else {
        push(Ep);
        push(cons(mkfix(Ip+1), Prog));
        Ep = closure_env(Acc);
        Prog = closure_prog(Acc);
    }
    return fixval(closure_ip(Acc));
}
```

The **conses()** function is similar to **length()**, but counts conses instead of list elements. The difference is subtle: **length()** expects a proper (i.e., **nil**-terminated) list as its argument, while **conses()** accepts any pair and counts the *cons cells* in the *spine* (see figure 23 on page 99) of its argument.

It is used to count the *fixed arguments* in the S-expression representation of an OSUBR or LSUBR.

```
int conses(cell n) {
    int     k;

    for (k = 0; pairp(n); n = cdr(n))
        k++;
    return k;
}
```

The **applis()** function implements the *arglis* auxiliary instruction (page 180) as well as the instructions *applis* and *applist*. **applis()** itself builds a partial stack frame (the *arguments* part, consisting of argument boxes and the argument count) from the list on the top of the stack ($S[Sp]$). The application itself is delegated to **apply()**. The *tail* flag is meaningless to **applis()** itself, it is just passed on to **apply()**.

```
int applis(int tail) {
    cell    a, p, new;
    int     k, i;

    a = boxref(stackref(Sp));
    if (!pairp(a) && a != NIL)
        error("apply: expected list", a);
    k = conses(a);
    stkalloc(k);
    Sp += k;
    i = Sp-1;
    for (p = a; p != NIL; p = cdr(p)) {
```

```
        if (atomp(p))
            error("apply: dotted list", a);
        new = box(car(p));
        stackset(i, new);
        i--;
    }
    new = mkfix(k);
    stackset(Sp, new);
    return apply(tail);
}
```

The **ret()** function implements the *return* instruction. It removes the stack frame of the most recently applied function, restores the *Fp* and *Ep* registers, and passes control back to the applying function.

```
int ret(void) {
    int     r, n;
    cell    *v;

    v = vector(Rts);
    Fp = fixval(v[Sp]);
    r = v[Sp-1];
    Prog = cdr(r);
    Ep = v[Sp-2];
    n = fixval(v[Sp-3]);
    Sp -= n+4;
    return fixval(car(r));
}
```

The **entcol()** function implements the *entcol* instruction (page 181). It collects optional arguments of OSUBRs and LSUBRs from the stack and saves them in a list which then replaces the first optional argument. When the number of arguments on the stack is smaller than the *fix* argument of **entcol()**, an error will be signalled.

```
void entcol(int fix) {
    int     n, na, i, s, d;
    cell    a, x, new;

    na = fixval(stackref(Sp-2));
    if (na < fix)
        error("too few arguments", UNDEF);
    protect(a = NIL);
    i = Sp-fix-3;
    for (n = na-fix; n; n--) {
        x = cons(boxref(stackref(i)), NIL);
        if (NIL == a) {
            a = x;
            car(Protected) = a;
        }
        else {
            cdr(a) = x;
            a = cdr(a);
        }
        i--;
    }
    a = unprot(1);
    if (na > fix) {
        new = box(a);
        stackset(Sp-fix-3, new);
    }
    else {
        push(NIL);
        s = Sp - na - 3;
        d = Sp - na - 2;
        for (i = na + 2; i >= 0; i--)
            stackset(d+i, stackref(s+i));
        new = mkfix(1+fix);
        stackset(Sp-2, new);
        new = box(NIL);
        stackset(Sp-fix-3, new);
    }
    push(mkfix(Fp));
```

```
    Fp = Sp-4;
}
```

17.1 Non-Local Exits

A *non-local exit* is the transfer of control to a different *context* or *stack frame*. A "local" transfer of control is performed by the *jmp*, *brf*, and *brt* instructions by changing the value of the *Ip* register to the desired location *in the same program fragment*. When transferring control to a different fragment, not only the *Prog* register has to change, but also the context of the program, consisting of the environment *Ep* and stack registers *Sp* and *Fp*, has to be adjusted.

Consider the following program:

```
(defun (find x a)
  (def exit nil)
  (defun (find2 a)
    (cond ((pair a) (or (find2 (car a))
                        (find2 (cdr a))))
          ((< x a)  (throw* exit a))
          (else     nil)))
  (catch*
    (lambda (ct)
      (setq exit ct)
      (find2 a))))

(find 5
 '(((1 . 2) . (3 . 4)) . ((5 . 6) . (7 . 8))))
```

It searches the tree *a* for a fixnum that is greater than *x* and returns it. When the search is not successful, the last clause of **cond** will return **nil**. The search itself is recursive, it attempts to find a suitable element in the car and cdr field of *a*. In case of success, the result is delivered to a catch tag outside of the *find2* function via **throw***, thereby bypassing the stack frames of all recursive applications of *find2*.

In the example, the program finds the element 6 which, given the order of traversal of *find2*, is the first element that is greater than 5.

When the element is found, there are (at least) five different contexts on the run time stack, labeled $C1$ through $C5$ in figure 50. $C1$ is the context of the *find* function itself, $C5$ is the context in which a matching element was found.

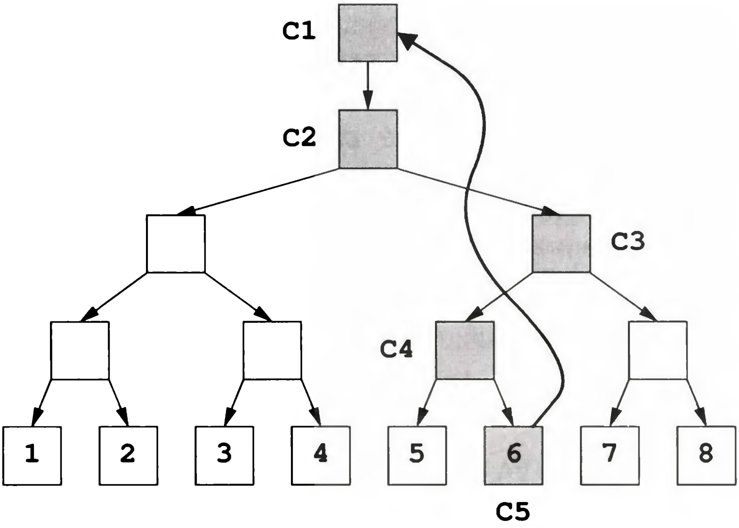


Fig. 50 – Non-local exit

The innermost context, which is on top of the stack, is $C5$. When returning from this context without throwing a catch tag, it will have to return to $C4$, which will then have to return to $C3$ and so on, all the way up to $C1$. The non-local transfer of control depicted by the bold arrow reinstates $C1$ immediately, thereby passing control to the point where the **catch*** function in that context returns.

Non-local exits as described here are lightweight – so lightweight in fact that they can be faster than just returning from a function multiple times. In the example discussed here they are used to speed up the computation. For another example, see the *ctak* function of the Gabriel benchmark suite [Gabriel1986].

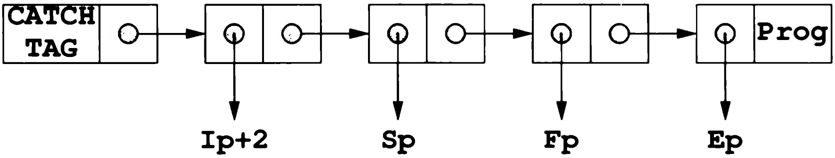


Fig. 51 – The structure of a catch tag object

The **mkctag()** function captures the *current context* (which is a subset of the *current continuation*), packages it in a *catch tag* and returns it. The address $Ip+2$ points to the instruction that will be interpreted when **catch*** returns. It is *two* instructions away, because *catch* is *always* followed by an *apply* instruction (see page 211). The structure of the catch tag object is shown in figure 51.

The **mkctag()** function implements half of the LISP9 function **catch*** (the other half being *apply*).

```
cell mkctag(void) {
    cell    n;

    n = cons(Ep, Prog);
    Tmp = n; n = cons(mkfix(Fp), n);
    Tmp = n; n = cons(mkfix(Sp), n);
    Tmp = n; n = cons(mkfix(Ip+2), n);
    Tmp = NIL;
    return mkatom(T_CATCHTAG, n);
}
```

The **throw()** function implements **throw***, which can be thought of as the inverse function of **catch***. It reinstates the context captured in the catch tag *ct* and and loads the accumulator with *v*, so that the application of **catch*** that saved the context will return *v*.

```
int throw(cell ct, cell v) {
    if (!ctagp(ct))
        expect("throw", "catch tag", ct);
    ct = cdr(ct);
    Ip = fixval(car(ct)); ct = cdr(ct);
    Sp = fixval(car(ct)); ct = cdr(ct);
    Fp = fixval(car(ct)); ct = cdr(ct);
    Ep = car(ct);         ct = cdr(ct);
    Prog = ct;
    Acc = v;
```

```
    return Ip;
}
```

The **throwerr()** function is used to catch errors using **catch-errors**. All it does is to extract the value to be returned by **catch-errors** from the environment variable ***errval*** and then throw the catch tag associated with the current error context. The mechanism will be discussed in detail in the LISP part of this book.

```
int throwerr(cell ct) {
    cell    n;

    n = assq(S_errval, Glob);
    n = NIL == n? NIL: cadr(n);
    return throw(ct, n);
}
```

Note that the catch/throw mechanism described here is a very portable and lightweight mechanism that is also known as *escape continuations* or *upward continuations*. Mechanisms like this are present in most programming languages. For instance, the C language provides it in the form of **setjmp()** and **longjmp()**.

A more general approach, as implemented by Scheme's **call-with-current-continuation** function [R4RS], allows to reify and reinstate the *current continuation*, which also captures the entire runtime stack, so that captured continuations can be reinstated from anywhere in a program. In particular, it would allow to transfer control *downward* in trees like the one in figure 50 on page 292.

To implement continuations properly, the runtime stack has to be copied when capturing *and* when reinstating a context, which is, of course, highly inefficient. Hence a number of techniques has been invented to make continuations more efficient, like "cactus stacks" [CHO1988] or "continuation passing style" [AIM349], [Appel1992].

BTW, the existence of the superficially peculiar construct

```
(catch* (lambda (tag) ...))
```

which passes the catch tag established by **catch*** to an anonymous function, has a specific reason. In early Scheme [AIM349], the catch construct had the form

```
(catch tag expr ...)
```

This works fine, but has one undesirable side effect: it introduces an additional *binding form* to the language, which complicates all kinds of programs that walk S-expressions and deduce bindings from binding forms. Without the original Scheme catch form, LISP has one binding construct, **lambda**, and all other binding forms are derived from **lambda**. So the **(catch tag ...)** form doubles the number of primitive binding constructs. This approach probably appeared first in RRS Scheme [AIM848] to implement **call-with-current-continuation**.

17.2 The Interpreter Loop

The variable **Run**, when set to a non-zero value, indicates that the abstract machine is running. Its value is changed to 0 when the user interrupts the computation in progress by sending an interrupt signal (**SIGINT**) to the LISP system. Because the signal is delivered asynchronously, **Run** is declared volatile.

The variable **Argv** is forward-declared. It will be bound to a list of command line arguments when the LISP9 system comes up.

```
volatile int    Run = 0;
cell            Argv;
```

The **run()** function implements the *interpreter loop* of the LISP9 abstract machine. It interprets the program fragment in the argument x and returns the result or *normal form* of the program in the Acc register. When the program is empty, the value of Acc defaults to **nil**. The value x is stored in the $Prog$ register and Ip is set to 0 before interpretation commences.

The **setjmp()** call at the beginning of the function handles *errors*

inside of the dynamic extent of **catch-errors** forms. Whenever an error condition is caught by **catch-errors**, the error handler (page 33) long-jumps back to this **setjmp()** call, where the context of an error handling function is reinstated by throwing the catch tag bound to the variable **Errtag**.

The rest of the function is one huge **switch** statement in an infinite loop. The loop is only ever left when the value of **Run** goes to 0. In this case, an error ("interrupted") will be signalled at the end of the function. The normal way for **run()** to terminate is by interpretation of a *halt* instruction, which will just return to the caller of **run()**.

Inside of the **switch** statement there is one **case** for each *instruction* and inlined *primitive function* of the LAM. These cases may be thought of as the *microcode* of the abstract machine. All microcode fragments use one of the following common patterns:

(1) Instructions that transfer control, like *jmp*, *brf*, or *throw*, simply change the *Ip* register and then go to the next instruction via a **break** statement. For instance, the implementation of *jmp* would be as simple as

```
Ip = op1(); break;
```

(2) All other instructions call **skip()** with the proper instruction format before continuing. Instructions with no inline operands call **skip(ISIZE0)**, instructions with one inline operand call **skip(ISIZE1)**, and instructions with two inline operands call **skip(ISIZE2)**. For instance, the zero-operand *drop* instruction would be implemented as

```
Sp--; skip(ISIZE0); break;
```

(3) Inline functions expect some of their arguments (operands) on the stack. They never have any inline operands. The first argument is always received in the *Acc* register and requires no cleanup. When two arguments are received, one has to be removed from the stack by calling **clear(1)**. When three arguments are received, two have to be cleaned up using **clear(2)**. For instance, the **eq** function would be implemented like this:

```
Acc = (Acc == arg(0))? TRUE: NIL;
clear(1);
```

```
skip(ISIZE0);
break;
```

(4) Some microcode fragments will perform *type checking* or *range checking*. For instance, the *negate* instruction looks like this:

```
if (!fixp(Acc)) expect("-", "fixnum", Acc);
if (INT_MIN == fixval(Acc))
    error("-: fixnum overflow", Acc);
if (fixval(Acc) < 0) Acc = mkfix(-fixval(Acc));
skip(ISIZE0);
break;
```

It makes sure that the argument in *Acc* has the expected type and falls into the range of the *negate* operation (**–INT_MIN** is undefined on two's complement machines). Only when these tests are passed, the result is computed.

(5) Many microcode fragments, especially those that implement more complex instructions, delegate their work to the functions described in chapter 16. For instance, *plus* is implemented as

```
Acc = add(Acc, arg(0));
clear(1);
skip(ISIZE0);
break;
```

One notable implementation is that of the *ref* instruction (look for **OP_REF** on the following page), because (a) it handles symbols that have been bound to an *undefined value* (see page 162) and (b) it traces references to symbols in the **Trace** buffer.

Otherwise there are no surprises on the following 20+ pages. Feel free to skim ahead! Things get more interesting again on page 318.

```
void run(cell x) {
    Acc = NIL;
    Prog = x;
    Ip = 0;
    if (setjmp(Errtag) != 0)
```

```
        Ip = throwerr(Handler);
    for (Run=1; Run;) switch (ins()) {
    case OP_APPLIS:
        Ip = applis(0);
        break;
    case OP_APPLIST:
        Ip = applis(1);
        break;
    case OP_TAILAPP:
        Ip = apply(1);
        break;
    case OP_APPLY:
        Ip = apply(0);
        break;
    case OP_QUOTE:
        Acc = vector(Obarray)[op1()];
        skip(ISIZE1);
        break;
    case OP_ARG:
        Acc = argref(op1());
        skip(ISIZE1);
        break;
    case OP_REF:
        Acc = boxref(envbox(op1()));
        if (UNDEF == Acc)
            error("undefined symbol",
                  vector(Symbols)[op2()]);
        if (Tp >= NTRACE) Tp = 0;
        Trace[Tp++] = op2();
        skip(ISIZE2);
        break;
    case OP_DROP:
        Sp--;
        skip(ISIZE0);
        break;
    case OP_POP:
        Acc = stackref(Sp);
        Sp--;
```

```
        skip(ISIZE0);
        break;
    case OP_PUSH:
        push(cons(Acc, NIL));
        skip(ISIZE0);
        break;
    case OP_PUSHTRUE:
        push(TRUE);
        skip(ISIZE0);
        break;
    case OP_PUSHVAL:
        push(mkfix(op1()));
        skip(ISIZE1);
        break;
    case OP_JMP:
        Ip = op1();
        break;
    case OP_BRF:
        if (NIL == Acc)
            Ip = op1();
        else
            skip(ISIZE1);
        break;
    case OP_BRT:
        if (NIL == Acc)
            skip(ISIZE1);
        else
            Ip = op1();
        break;
    case OP_HALT:
        return;
    case OP_CATCHSTAR:
        push(box(mkctag()));
        push(mkfix(1));
        skip(ISIZE0);
        break;
    case OP_THROWSTAR:
        Ip = throw(Acc, arg(0));
```

```
        break;
    case OP_MKENV:
        Acc = mkvec(op1());
        skip(ISIZE1);
        break;
    case OP_PROPENV:
        Acc = Ep;
        skip(ISIZE0);
        break;
    case OP_CPARG:
        vector(Acc)[op2()] = argbox(op1());
        skip(ISIZE2);
        break;
    case OP_CPREF:
        vector(Acc)[op2()] = envbox(op1());
        skip(ISIZE2);
        break;
    case OP_CLOSURE:
        Acc = closure(op1(), Acc);
        skip(ISIZE1);
        break;
    case OP_ENTER:
        if (fixval(stackref(Sp-2)) != op1())
            error("wrong number of arguments",
                  UNDEF);
        push(mkfix(Fp));
        Fp = Sp-4;
        skip(ISIZE1);
        break;
    case OP_ENTCOL:
        entcol(op1());
        skip(ISIZE1);
        break;
    case OP_RETURN:
        Ip = ret();
        break;
    case OP_SETARG:
        boxset(argbox(op1()), Acc);
```

```
        skip(ISIZE1);
        break;
    case OP_SETREF:
        boxset(envbox(op1()), Acc);
        skip(ISIZE1);
        break;
    case OP_MACRO:
        newmacro(op1(), Acc);
        skip(ISIZE1);
        break;
    case OP_CMDLINE:
        Acc = Argv;
        skip(ISIZE0);
        break;
    case OP_QUIT:
        exit(EXIT_SUCCESS);
        skip(ISIZE0);
        break;
    case OP_OBTAB:
        Acc = Obarray;
        skip(ISIZE0);
        break;
    case OP_SYMTAB:
        Acc = Symbols;
        skip(ISIZE0);
        break;
    case OP_ERROR:
        if (!stringp(Acc))
            expect("error", "string", Acc);
        error((char *) string(Acc), UNDEF);
        skip(ISIZE0);
        break;
    case OP_ERROR2:
        if (!stringp(Acc))
            expect("error", "string", Acc);
        error((char *) string(Acc), arg(0));
        clear(1);
        skip(ISIZE0);
```

```
        break;
    case OP_ERRPORT:
        Acc = mkport(Errport, T_OUTPORT);
        skip(ISIZE0);
        break;
    case OP_INPORT:
        Acc = mkport(Inport, T_INPORT);
        skip(ISIZE0);
        break;
    case OP_OUTPORT:
        Acc = mkport(Outport, T_OUTPORT);
        skip(ISIZE0);
        break;
    case OP_GC:
        Acc = b_gc();
        skip(ISIZE0);
        break;
    case OP_GENSYM:
        Acc = gensym();
        skip(ISIZE0);
        break;
    case OP_ABS:
        if (!fixp(Acc))
            expect("abs", "fixnum", Acc);
        if (INT_MIN == fixval(Acc))
            error("abs: fixnum overflow", Acc);
        if (fixval(Acc) < 0)
            Acc = mkfix(-fixval(Acc));
        skip(ISIZE0);
        break;
    case OP_ALPHAC:
        if (!charp(Acc))
            expect("alphac", "char", Acc);
        Acc = isalpha(charval(Acc))? TRUE: NIL;
        skip(ISIZE0);
        break;
    case OP_ATOM:
        Acc = pairp(Acc)? NIL: TRUE;
```

```
        skip(ISIZE0);
        break;
    case OP_CAR:
        if (!pairp(Acc))
            expect("car", "pair", Acc);
        Acc = car(Acc);
        skip(ISIZE0);
        break;
    case OP_CDR:
        if (!pairp(Acc))
            expect("cdr", "pair", Acc);
        Acc = cdr(Acc);
        skip(ISIZE0);
        break;
    case OP_CAAR:
        if (!pairp(Acc) || !pairp(car(Acc)))
            expect("caar", "nested pair", Acc);
        Acc = caar(Acc);
        skip(ISIZE0);
        break;
    case OP_CADR:
        if (!pairp(Acc) || !pairp(cdr(Acc)))
            expect("cadr", "nested pair", Acc);
        Acc = cadr(Acc);
        skip(ISIZE0);
        break;
    case OP_CDAR:
        if (!pairp(Acc) || !pairp(car(Acc)))
            expect("cdar", "nested pair", Acc);
        Acc = cdar(Acc);
        skip(ISIZE0);
        break;
    case OP_CDDR:
        if (!pairp(Acc) || !pairp(cdr(Acc)))
            expect("cddr", "nested pair", Acc);
        Acc = cddr(Acc);
        skip(ISIZE0);
        break;
```

```
case OP_CHAR:
    if (!fixp(Acc))
        expect("char", "fixnum", Acc);
    if (fixval(Acc) < 0 || fixval(Acc) > 255)
        error("char: value out of range",
              Acc);
    Acc = mkchar(fixval(Acc));
    skip(ISIZE0);
    break;
case OP_CHARP:
    Acc = charp(Acc)? TRUE: NIL;
    skip(ISIZE0);
    break;
case OP_CHARVAL:
    if (!charp(Acc))
        expect("charval", "char", Acc);
    Acc = mkfix(charval(Acc));
    skip(ISIZE0);
    break;
case OP_CLOSE_PORT:
    if (!inportp(Acc) && !outportp(Acc))
        expect("close-port", "port", Acc);
    close_port(portno(Acc));
    Acc = NIL;
    skip(ISIZE0);
    break;
case OP_CONSTP:
    Acc = constp(Acc)? TRUE: NIL;
    skip(ISIZE0);
    break;
case OP_CTAGP:
    Acc = ctagp(Acc)? TRUE: NIL;
    skip(ISIZE0);
    break;
case OP_DELETE:
    if (!stringp(Acc))
        expect("delete", "string", Acc);
    if (remove((char *) string(Acc)) < 0)
```

```
            error("delete: cannot delete", Acc);
        Acc = NIL;
        skip(ISIZE0);
        break;
    case OP_DOWNCASE:
        if (!charp(Acc))
            expect("downcase", "char", Acc);
        Acc = mkchar(tolower(charval(Acc)));
        skip(ISIZE0);
        break;
    case OP_DUMP_IMAGE:
        if (!stringp(Acc))
            expect("dump-image", "string", Acc);
        dump_image(Acc);
        Acc = TRUE;
        skip(ISIZE0);
        break;
    case OP_EOFP:
        Acc = (EOFMARK == Acc? TRUE: NIL);
        skip(ISIZE0);
        break;
    case OP_EVAL:
        Acc = eval(Acc, 1);
        skip(ISIZE0);
        break;
    case OP_EXISTSP:
        if (!stringp(Acc))
            expect("existsp", "string", Acc);
        Acc = existsp((char *) string(Acc));
        skip(ISIZE0);
        break;
    case OP_FIXP:
        Acc = fixp(Acc)? TRUE: NIL;
        skip(ISIZE0);
        break;
    case OP_FORMAT:
        Acc = format(Acc);
        skip(ISIZE0);
```

```
        break;
    case OP_FUNP:
        Acc = closurep(Acc)? TRUE: NIL;
        skip(ISIZE0);
        break;
    case OP_INPORTP:
        Acc = inportp(Acc)? TRUE: NIL;
        skip(ISIZE0);
        break;
    case OP_LISTSTR:
        if (!listp(Acc))
            expect("liststr", "list", Acc);
        Acc = liststr(Acc);
        skip(ISIZE0);
        break;
    case OP_LISTVEC:
        if (!listp(Acc))
            expect("listvec", "list", Acc);
        Acc = listvec(Acc, 0);
        skip(ISIZE0);
        break;
    case OP_LOAD:
        load(Acc);
        Acc = TRUE;
        skip(ISIZE0);
        break;
    case OP_LOWERC:
        if (!charp(Acc))
            expect("lowerc", "char", Acc);
        Acc = islower(charval(Acc))? TRUE: NIL;
        skip(ISIZE0);
        break;
    case OP_MX:
        Acc = expand(Acc, 1);
        skip(ISIZE0);
        break;
    case OP_MX1:
        Acc = expand(Acc, 0);
```

```
            skip(ISIZE0);
            break;
        case OP_NEGATE:
            if (!fixp(Acc))
                expect("-", "fixnum", Acc);
            if (INT_MIN == fixval(Acc))
                error("-: fixnum overflow", Acc);
            Acc = mkfix(-fixval(Acc));
            skip(ISIZE0);
            break;
        case OP_NULL:
            Acc = (NIL == Acc? TRUE: NIL);
            skip(ISIZE0);
            break;
        case OP_NUMSTR:
            if (!fixp(Acc))
                expect("numstr", "fixnum", Acc);
            if (!fixp(arg(0)))
                expect("numstr", "fixnum", arg(0));
            Acc = numstr(Acc, fixval(arg(0)));
            clear(1);
            skip(ISIZE0);
            break;
        case OP_NUMERIC:
            if (!charp(Acc))
                expect("numeric", "char", Acc);
            Acc = isdigit(charval(Acc))? TRUE: NIL;
            skip(ISIZE0);
            break;
        case OP_OPEN_INFILE:
            if (!stringp(Acc))
                expect("open-infile", "string", Acc);
            Acc = openfile(Acc, 0);
            skip(ISIZE0);
            break;
        case OP_OPEN_OUTFILE:
            if (!stringp(Acc))
                expect("open-outfile", "string", Acc);
```

```
        Acc = openfile(Acc, NIL == arg(0)? 1: 2);
        clear(1);
        skip(ISIZE0);
        break;
    case OP_OUTPORTP:
        Acc = outportp(Acc)? TRUE: NIL;
        skip(ISIZE0);
        break;
    case OP_PAIR:
        Acc = pairp(Acc)? TRUE: NIL;
        skip(ISIZE0);
        break;
    case OP_PEEKC:
        if (!inportp(Acc))
            expect("peekc", "inport", Acc);
        Acc = b_readc(portno(Acc), 1);
        skip(ISIZE0);
        break;
    case OP_READ:
        if (!inportp(Acc) && !stringp(Acc))
            expect("read", "inport", Acc);
        Acc = b_read(Acc);
        skip(ISIZE0);
        break;
    case OP_READC:
        if (!inportp(Acc))
            expect("readc", "inport", Acc);
        Acc = b_readc(portno(Acc), 0);
        skip(ISIZE0);
        break;
    case OP_CONC:
        Acc = lconc(Acc);
        skip(ISIZE0);
        break;
    case OP_NCONC:
        Acc = nlconc(Acc);
        skip(ISIZE0);
        break;
```

```
    case OP_SCONC:
        Acc = sconc(Acc);
        skip(ISIZE0);
        break;
    case OP_SET_INPORT:
        if (!inportp(Acc))
            expect("set-inport", "inport", Acc);
        Inport = portno(Acc);
        skip(ISIZE0);
        break;
    case OP_SET_OUTPORT:
        if (!outportp(Acc))
            expect("set-outport", "outport", Acc);
        Outport = portno(Acc);
        skip(ISIZE0);
        break;
    case OP_SSIZE:
        if (!stringp(Acc))
            expect("ssize", "string", Acc);
        Acc = mkfix(stringlen(Acc)-1);
        skip(ISIZE0);
        break;
    case OP_STRNUM:
        if (!stringp(Acc))
            expect("strnum", "string", Acc);
        if (!fixp(arg(0)))
            expect("strnum", "fixnum", arg(0));
        Acc = strnum((char *) string(Acc),
                        fixval(arg(0)));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_SYMBOLP:
        Acc = symbolp(Acc)? TRUE: NIL;
        skip(ISIZE0);
        break;
    case OP_SYMBOL:
        if (!stringp(Acc))
```

```
            expect("symbol", "string", Acc);
        Acc = b_symbol(Acc);
        skip(ISIZE0);
        break;
    case OP_SYMNAME:
        if (!symbolp(Acc))
            expect("symname", "symbol", Acc);
        Acc = b_symname(Acc);
        skip(ISIZE0);
        break;
    case OP_STRINGP:
        Acc = stringp(Acc)? TRUE: NIL;
        skip(ISIZE0);
        break;
    case OP_STRLIST:
        if (!stringp(Acc))
            expect("strlist", "string", Acc);
        Acc = strlist(Acc);
        skip(ISIZE0);
        break;
    case OP_SYSCMD:
        if (!stringp(Acc))
            expect("syscmd", "string", Acc);
        Acc = mkfix(system((char *) string(Acc))
                    >> 8);
        skip(ISIZE0);
        break;
    case OP_UNTAG:
        Acc = untag(Acc);
        skip(ISIZE0);
        break;
    case OP_UPCASE:
        if (!charp(Acc))
            expect("upcase", "char", Acc);
        Acc = mkchar(toupper(charval(Acc)));
        skip(ISIZE0);
        break;
    case OP_UPPERC:
```

```
        if (!charp(Acc))
            expect("upperc", "char", Acc);
        Acc = isupper(charval(Acc))? TRUE: NIL;
        skip(ISIZE0);
        break;
    case OP_VCONC:
        Acc = vconc(Acc);
        skip(ISIZE0);
        break;
    case OP_VECLIST:
        if (!vectorp(Acc))
            expect("veclist", "vector", Acc);
        Acc = veclist(Acc);
        skip(ISIZE0);
        break;
    case OP_VECTORP:
        Acc = vectorp(Acc)? TRUE: NIL;
        skip(ISIZE0);
        break;
    case OP_VSIZE:
        if (!vectorp(Acc))
            expect("vsize", "vector", Acc);
        Acc = mkfix(veclen(Acc));
        skip(ISIZE0);
        break;
    case OP_WHITEC:
        if (!charp(Acc))
            expect("whitec", "char", Acc);
        Acc = whitespc(charval(Acc))? TRUE: NIL;
        skip(ISIZE0);
        break;
    case OP_BITOP:
        Acc = bitop(Acc, arg(0), arg(1));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_CLESS:
        cless(Acc, arg(0));
```

```
        clear(1);
        skip(ISIZE0);
        break;
    case OP_CLTEQ:
        clteq(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_CEQUAL:
        cequal(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_CGRTR:
        cgrtr(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_CGTEQ:
        cgteq(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_CONS:
        Acc = cons(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_DIV:
        Acc = intdiv(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_EQ:
        Acc = (Acc == arg(0))? TRUE: NIL;
        clear(1);
        skip(ISIZE0);
        break;
```

```
    case OP_EQUAL:
        equal(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_GRTR:
        grtr(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_GTEQ:
        gteq(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_LESS:
        less(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_LTEQ:
        lteq(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_MAX:
        if (fixval(arg(0)) > fixval(Acc))
            Acc = arg(0);
        clear(1);
        skip(ISIZE0);
        break;
    case OP_MIN:
        if (fixval(arg(0)) < fixval(Acc))
            Acc = arg(0);
        clear(1);
        skip(ISIZE0);
        break;
    case OP_MINUS:
```

```
        Acc = xsub(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_MKSTR:
        Acc = b_mkstr(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_MKVEC:
        Acc = b_mkvec(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_NRECONC:
        if (!listp(Acc))
            expect("nreconc", "list", Acc);
        if (constp(Acc))
            error("nreconc: immutable", Acc);
        Acc = nreconc(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_PLUS:
        Acc = add(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_PRIN:
        if (!outportp(arg(0)))
            expect("prin", "outport", arg(0));
        b_prin(Acc, portno(arg(0)), 1);
        clear(1);
        skip(ISIZE0);
        break;
    case OP_PRINC:
        if (!outportp(arg(0)))
            expect("princ", "outport", arg(0));
```

```
        b_prin(Acc, portno(arg(0)), 0);
        clear(1);
        skip(ISIZE0);
        break;
    case OP_RECONC:
        if (!listp(Acc))
            expect("reconc", "list", Acc);
        Acc = reconc(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_REM:
        Acc = intrem(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_SETCAR:
        if (!pairp(Acc))
             expect("setcar", "pair", Acc);
        if (constp(Acc))
            error("setcar: immutable", Acc);
        car(Acc) = arg(0);
        clear(1);
        skip(ISIZE0);
        break;
    case OP_SETCDR:
        if (!pairp(Acc))
            expect("setcdr", "pair", Acc);
        if (constp(Acc))
            error("setcdr: immutable", Acc);
        cdr(Acc) = arg(0);
        clear(1);
        skip(ISIZE0);
        break;
    case OP_SLESS:
        Acc = sless(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
```

```
            break;
        case OP_SLTEQ:
            Acc = slteq(Acc, arg(0));
            clear(1);
            skip(ISIZE0);
            break;
        case OP_SEQUAL:
            Acc = sequal(Acc, arg(0));
            clear(1);
            skip(ISIZE0);
            break;
        case OP_SGRTR:
            Acc = sgrtr(Acc, arg(0));
            clear(1);
            skip(ISIZE0);
            break;
        case OP_SGTEQ:
            Acc = sgteq(Acc, arg(0));
            clear(1);
            skip(ISIZE0);
            break;
        case OP_SILESS:
            Acc = siless(Acc, arg(0));
            clear(1);
            skip(ISIZE0);
            break;
        case OP_SILTEQ:
            Acc = silteq(Acc, arg(0));
            clear(1);
            skip(ISIZE0);
            break;
        case OP_SIEQUAL:
            Acc = siequal(Acc, arg(0));
            clear(1);
            skip(ISIZE0);
            break;
        case OP_SIGRTR:
            Acc = sigrtr(Acc, arg(0));
```

```
        clear(1);
        skip(ISIZE0);
        break;
    case OP_SIGTEQ:
        Acc = sigteq(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_SFILL:
        sfill(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_SREF:
        Acc = sref(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_SSET:
        sset(Acc, arg(0), arg(1));
        clear(2);
        skip(ISIZE0);
        break;
    case OP_SUBSTR:
        Acc = substr(Acc, arg(0), arg(1));
        clear(2);
        skip(ISIZE0);
        break;
    case OP_SUBVEC:
        Acc = subvec(Acc, arg(0), arg(1));
        clear(2);
        skip(ISIZE0);
        break;
    case OP_TIMES:
        Acc = mul(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
```

```
    case OP_VFILL:
        vfill(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_VREF:
        Acc = vref(Acc, arg(0));
        clear(1);
        skip(ISIZE0);
        break;
    case OP_VSET:
        vset(Acc, arg(0), arg(1));
        clear(2);
        skip(ISIZE0);
        break;
    case OP_WRITEC:
        if (!charp(Acc))
            expect("writec", "char", Acc);
        if (!outportp(arg(0)))
            expect("writec", "outport", arg(0));
        b_writec(charval(Acc), portno(arg(0)));
        clear(1);
        skip(ISIZE0);
        break;
    default:
        error("illegal instruction",
                mkfix(ins()));
        return;
    }
    error("interrupted", UNDEF);
}
```

17.3 EVAL

This section implements the **eval** function, which macro-expands, compiles, and interprets LISP programs. When a program is *purely functional*, **eval** returns its *result* or *normal form*. Of course, **eval** does always return *some* result, but if a program has an

effect that cannot be explained in terms of a functional mapping, then the result may be incomplete or even uninteresting. For instance, the program

```
(cons 'a (cons 'b (cons 'c nil)))
```

is purely functional and evaluates to the normal form **(a b c)**. The program.

```
(prin (+ 1 2))
```

has the result 3, but also the effect of printing that result. The program

```
(print (+ 1 2))
```

prints the same value as the above program, but has an unspecific result, which is therefore uninteresting.

The individual steps in the process of evaluation have been implemented in the code so far. The **interpret()** function is a final building stone in the last phase of expression evaluation. It *interprets* the bytecode program x and returns its result. It will interpret the bytecode in the current *top level environment*, that is, it will set up the Ep register to point to $E0$ before running the bytecode program. It will also construct $E0$ from the deep-binding environment **Glob**.

The top level environment $E0$ has to be created freshly each time **interpret()** is called, because new bindings may have been added during closure conversion, and these new bindings have to be added to $E0$, too.

The deep binding environment and the (shallow-binding) $E0$ environment share *the same locations*. Hence modifying values stored in one will also modify the other. This is necessary for special variables, like ** and ***imagefile***, to work properly, because they are modified by code outside of the LAM, but interpreted by LAM code. Figure 52 illustrates the equivalence of the two environments.

The deep-binding environment allows the LISP9 system to refer to bindings by name, using the **assq()** function, while the shallow-binding $E0$ environment implements constant-time lookup of

values. They can be considered to be different interfaces for accessing the same bindings.

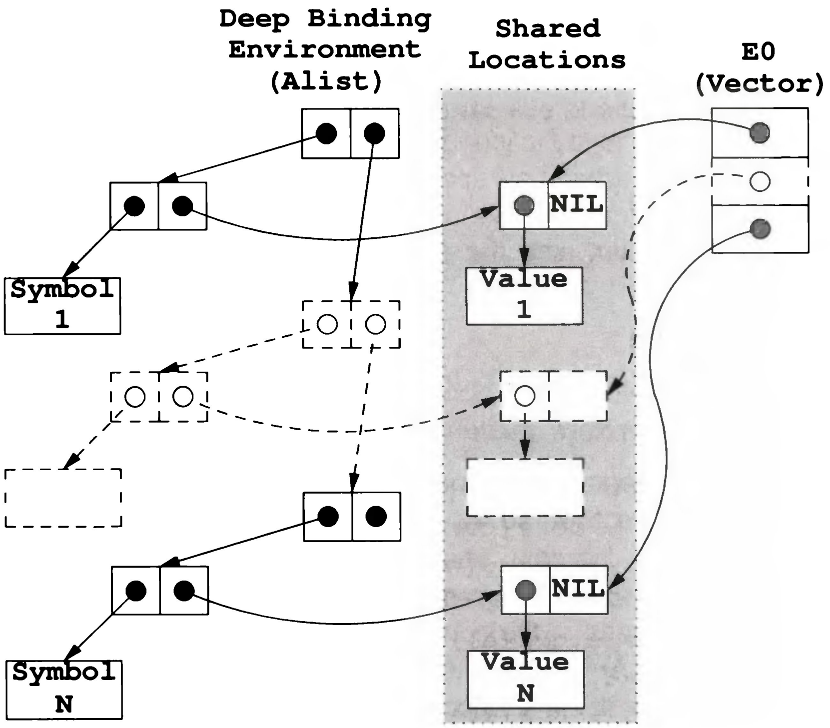


Fig. 52 – Two environments sharing the same bindings

```
cell interpret(cell x) {
    cell    n;
    int     i;

    E0 = mkvec(length(Glob));
    i = 0;
    for (n = Glob; n != NIL; n = cdr(n)) {
        vector(E0)[i] = cdar(n);
        i++;
    }
    Ep = E0;
    run(x);
    return Acc;
}
```

LISP functions like **eval** and **load** as well as the entire process of macro expansion rely on the **eval()** function being reentrant. The **begin_rec()** ("begin recursion") and **end_rec()** ("end recursion") functions are used to save and restore the global state of **eval()**. This state basically consists of the abstract machine registers, except for Acc, which is used to return the result of an evaluation and $E0$, which is constant during the evaluation of an expression.

The state is saved on the GC protection stack, which in this case serves as a general purpose stack outside of the LAM.

```
void begin_rec(void) {
    protect(Prog);
    protect(Ep);
    protect(mkfix(Ip));
    protect(mkfix(Sp));
    protect(mkfix(Fp));
}

void end_rec(void) {
    Fp = fixval(unprot(1));
    Sp = fixval(unprot(1));
    Ip = fixval(unprot(1));
    Ep = unprot(1);
    Prog = unprot(1);
}
```

With all the other pieces already in place, **eval()** is a simple function. It saves its argument x on the stack, because it might be an unprotected expression, like one generated by the macro expander. When the r argument is non-zero, it also saves the LAM state and restores it later, thereby becoming reentrant.

The process of evaluation is straightforward:

- macro-expand x, giving x
- syntax-check x

- closure-convert x, giving x
- compile x to LAM code, giving x
- interpret x, giving the final result x

Syntax checking is performed *after* macro-expansion, because the number of special forms will be reduced significantly during macro-expansion. Derived special operators will do their own syntax checking.

Syntax checking is an identity function with error reporting as a conditional effect, so its value can be ignored.

```
cell eval(cell x, int r) {
    Tmp = x;
    if (r) begin_rec();
    protect(x);
    Tmp = NIL;
    x = expand(x, 1);   car(Protected) = x;
    syncheck(x, 1, 1);
    x = clsconv(x);     car(Protected) = x;
    x = compile(x);     car(Protected) = x;
    x = interpret(x);
    unprot(1);
    if (r) end_rec();
    return x;
}
```

---

18. The REPL

This short chapter implements the *REPL*, the *read eval print loop*, which is the interactive interface to the LISP system. As its name indicates, it

- reads an expression from its input source, typically a terminal
- evaluates the expression
- prints the value of the expression (if any)
- loops, i.e. starts from the beginning

Indeed, a very simple incarnation of the REPL would look like this:

```
(let loop ()
  (print (eval (read)))
  (loop))
```

It looks neat, because it evaluates its expression in the order **read**, **eval**, **print**, **loop**, as indicated by the acronym REPL. However, it does not handle errors, aborted input, input redirection, etc. This is why the actual code tends to be a bit more complicated.

The variable **Intr** is set by the **kbdintr()** ("keyboard interrupt") function, whenever a keyboard interrupt (**SIGINT**) signal is received. When it is set to a non-zero value, the reader will stop reading further input and the printer will stop printing.

Various interrupt conditions are checked at safe points in the reader, printer, macro expander, and interpreter loop of the abstract machine, because processing interrupt signals asynchronously could leave the system in an undefined state, especially when interrupting the garbage collector.

The variable **Run** is part of the abstract machine. Setting it to zero will stop the machine.

The variable **Mxlev** is part of the macro expander. Setting it to -1 will stop macro expansion.

The **kbdint()** function does all of the above.

```
volatile int     Intr = 0;

void kbdintr(int sig) {
    Run = 0;
    Intr = 1;
    Mxlev = -1;
}
```

The **initrts()** ("initialize runtime stack") function allocates a *runtime stack* with an initial size of **CHUNKSIZE** slots and resets the stack pointer and frame pointer so that the stack will be empty.

The variable **Rts**, which is bound to the stack, is set to NIL before allocating the stack in order to signal the garbage collector that no runtime stack exists. Omitting this step would make the garbage collector trace whatever **Rts** is bound to.

```
void initrts(void) {
    Rts = NIL;
    Rts = mkvec(CHUNKSIZE);
    Sz = CHUNKSIZE;
    Sp = -1;
    Fp = -1;
}
```

The **repl()** function implements the LISP9 REPL. The variable **Quiet**, when set to a non-zero value, will change the behavior of the interface as follows:

- uncaught errors will exit the LISP system
- keyboard interrupts will exit the system
- no prompts will print
- no newline will print when exiting normally

A newline is printed upon exit in non-quiet mode, because otherwise the shell prompt of a Unix system may print on the same line as the last prompt of the LISP system, which may look confusing. In batch mode, however, that newline character would

be extra output, which may disturb pipe processing.

The **setjmp()** call at the beginning of the function catches all errors that do not occur inside of a user-defined error context. Its effect is to simply reenter the REPL.

Before reading an expression from the input source, the LISP9 REPL resets the standard I/O ports, clears the trace buffer, and creates a fresh runtime stack. Because the runtime stack may grow during evaluation, a fresh, small stack is allocated in each iteration. The GC protection stack is also cleared.

When an EOF marker is read *and* the **Intr** flag is still *cleared*, the REPL will exit. On some systems, interrupting I/O seems to generate an EOF marker, so checking the **Intr** flag stops the system from exiting erroneously in this case.

The **Mxlev** variable is initialized with 0 inside of the REPL, because initializing it at any other point would not work due to the recursive nature of **eval**.

After evaluation, the value delivered by **eval** is stored in the location bound to the variable **. This point in the REPL is only reached when no error occurred anywhere in the process, so erroneous results will never be stored.

```
int     Quiet = 0;

void repl(void) {
    cell    x;

    if (setjmp(Restart) && Quiet)
        exit(EXIT_FAILURE);
    if (!Quiet) signal(SIGINT, kbdintr);
    for (;;) {
        reset_stdports();
        clrtrace();
        initrts();
        Protected = NIL;
        Run = 0;
        Intr = 0;
```

```
        if (!Quiet) {
            prints("* ");
            flush();
        }
        x = xread();
        if (EOFMARK == x && !Intr) break;
        Mxlev = 0;
        x = eval(x, 0);
        bindset(S_starstar, x);
        print(x);
    }
    if (!Quiet) nl();
}
```

19. Startup and Initialization

There is a lot of state in the LISP9 system, and the **init()** function initializes it.

The first part sets up the **Ports** array and the standard I/O ports. The ports are locked, so they cannot be recycled by the garbage collector.

The next step is to allocate and initialize the node and vector pools. Initialization is performed by running a full (node and vector pool) GC.

An initial runtime stack is allocated and the trace buffer is cleared. Normally, this is done in the REPL, but the REPL is never reached when a program is run in batch mode, so these steps are included here, too.

A very large part of **init()** deals with allocating all kinds of objects: predefined data objects, like **Blank**, **Zero**, **Nullvec**, etc, as well as the initial *symbol table* and *object table*.

All the predefined symbols of the LISP9 system will also be created here. There is quite a lot of them, so feel free to skip ahead!

Do not miss the end of the function, though, where the *special variables* of the system (like **, ***imagefile***, and ***errtag***) will be set up.

```
void init(void) {
    int     i;

    for (i=2; i<NPORTS; i++) Ports[i] = NULL;
    Ports[0] = stdin;  Port_flags[0] = LOCK_TAG;
    Ports[1] = stdout; Port_flags[1] = LOCK_TAG;
    Ports[2] = stderr; Port_flags[2] = LOCK_TAG;
    alloc_nodepool();
    alloc_vecpool();
    gcv();
```

```
initrts();
clrtrace();
Nullvec = newvec(T_VECTOR, 0);
Nullstr = newvec(T_STRING, 1);
Blank = mkchar(' ');
Zero = mkfix(0);
One = mkfix(1);
Ten = mkfix(10);
Symbols = mkvec(CHUNKSIZE);
Symhash = mkht(CHUNKSIZE);
Obhash = mkht(CHUNKSIZE);
Obarray = mkvec(CHUNKSIZE);
Obmap = mkstr("", CHUNKSIZE);
memset(string(Obmap), OBFREE, CHUNKSIZE);
symref("?");
I_a = symref("a");
I_e = symref("e");
I_arg = symref("%arg");
I_closure = symref("%closure");
I_ref = symref("%ref");
S_apply = symref("apply");
S_comment = symref("--");
S_def = symref("def");
S_defmac = symref("defmac");
S_defun = symref("defun");
S_errtag = symref("*errtag*");
S_errval = symref("*errval*");
S_if = symref("if");
S_ifstar = symref("if*");
S_imagefile = symref("*imagefile*");
S_labels = symref("labels");
S_lambda = symref("lambda");
S_macro = symref("macro");
S_prog = symref("prog");
S_quiet = symref("*quiet*");
S_quote = symref("quote");
S_qquote = symref("qquote");
S_unquote = symref("unquote");
```

```
S_splice = symref("splice");
S_starstar = symref("**");
S_setq = symref("setq");
S_start = symref("start");
P_abs = symref("abs");
P_alphac = symref("alphac");
P_atom = symref("atom");
P_bitop = symref("bitop");
P_caar = symref("caar");
P_cadr = symref("cadr");
P_car = symref("car");
P_catchstar = symref("catch*");
P_cdar = symref("cdar");
P_cddr = symref("cddr");
P_cdr = symref("cdr");
P_cequal = symref("c=");
P_cgrtr = symref("c>");
P_cgteq = symref("c>=");
P_char = symref("char");
P_charp = symref("charp");
P_charval = symref("charval");
P_cless = symref("c<");
P_close_port = symref("close-port");
P_clteq = symref("c<=");
P_cmdline = symref("cmdline");
P_conc = symref("conc");
P_cons = symref("cons");
P_constp = symref("constp");
P_ctagp = symref("ctagp");
P_delete = symref("delete");
P_div = symref("div");
P_downcase = symref("downcase");
P_dump_image = symref("dump-image");
P_eofp = symref("eofp");
P_eq = symref("eq");
P_equal = symref("=");
P_error = symref("error");
P_errport = symref("errport");
```

```
P_eval = symref("eval");
P_existsp = symref("existsp");
P_fixp = symref("fixp");
P_format = symref("format");
P_funp = symref("funp");
P_gc = symref("gc");
P_gensym = symref("gensym");
P_grtr = symref(">");
P_gteq = symref(">=");
P_inport = symref("inport");
P_inportp = symref("inportp");
P_less = symref("<");
P_liststr = symref("liststr");
P_listvec = symref("listvec");
P_load = symref("load");
P_lowerc = symref("lowerc");
P_lteq = symref("<=");
P_max = symref("max");
P_min = symref("min");
P_minus = symref("-");
P_mkstr = symref("mkstr");
P_mkvec = symref("mkvec");
P_mx = symref("mx");
P_mx1 = symref("mx1");
P_not = symref("not");
P_nconc = symref("nconc");
P_nreconc = symref("nreconc");
P_null = symref("null");
P_numeric = symref("numeric");
P_numstr = symref("numstr");
P_obtab = symref("obtab");
P_open_infile = symref("open-infile");
P_open_outfile = symref("open-outfile");
P_outport = symref("outport");
P_outportp = symref("outportp");
P_pair = symref("pair");
P_peekc = symref("peekc");
P_plus = symref("+");
```

```
P_prin = symref("prin");
P_princ = symref("princ");
P_quit = symref("quit");
P_read = symref("read");
P_readc = symref("readc");
P_reconc = symref("reconc");
P_rem = symref("rem");
P_sconc = symref("sconc");
P_sequal = symref("s=");
P_set_inport = symref("set-inport");
P_set_outport = symref("set-outport");
P_setcar = symref("setcar");
P_setcdr = symref("setcdr");
P_sfill = symref("sfill");
P_sgrtr = symref("s>");
P_sgteq = symref("s>=");
P_siequal = symref("si=");
P_sigrtr = symref("si>");
P_sigteq = symref("si>=");
P_siless = symref("si<");
P_silteq = symref("si<=");
P_sless = symref("s<");
P_slteq = symref("s<=");
P_sref = symref("sref");
P_sset = symref("sset");
P_ssize = symref("ssize");
P_stringp = symref("stringp");
P_strlist = symref("strlist");
P_strnum = symref("strnum");
P_substr = symref("substr");
P_subvec = symref("subvec");
P_symbol = symref("symbol");
P_symbolp = symref("symbolp");
P_symname = symref("symname");
P_symtab = symref("symtab");
P_syscmd = symref("syscmd");
P_throwstar = symref("throw*");
P_times = symref("*");
```

```
    P_untag = symref("untag");
    P_upcase = symref("upcase");
    P_upperc = symref("upperc");
    P_vconc = symref("vconc");
    P_veclist = symref("veclist");
    P_vectorp = symref("vectorp");
    P_vfill = symref("vfill");
    P_vref = symref("vref");
    P_vset = symref("vset");
    P_vsize = symref("vsize");
    P_whitec = symref("whitec");
    P_writec = symref("writec");
    bindnew(S_errtag, NIL);
    bindnew(S_errval, NIL);
    bindnew(S_imagefile, NIL);
    bindnew(S_quiet, NIL);
    bindnew(S_starstar, NIL);
    bindnew(S_start, NIL);
}
```

The **start()** function evaluates

```
(start)
```

when the variable **start** is bound to a closure. This mechanism can be used to perform user-defined actions when the LISP9 system comes up. For instance, the following function, when entered and saved to the default LISP9 image, would report the number of free nodes and vcells:

```
(defun (start)
  (cond ((not *quiet*)
          (let ((g (gc)))
            (terpri)
            (princ (car g))
            (princ " NODES\n")
            (princ (cadr g))
            (princ " VCELLS\n\n")))))
```

```
void start(void) {
    cell    n;

    n = assq(S_start, Glob);
    if (NIL == n || closurep(cadr(n)) == 0)
        return;
    n = cons(cadr(n), NIL);
    eval(n, 0);
}
```

An *image variable* is an internal variable of the LISP system that needs to be stored in an image file, because its value is needed to resume operation when the image is being loaded. In the case of LISP9, these are

- the freelist and free vector pointer
- the variables that constitute the symbol table
- the variables that constitute the object table
- the runtime stack
- the (deep-binding) top level environment
- the environment binding macro names to macros

It might not be obvious why the *runtime stack* has to be included in the image variables, because it is created dynamically during initialization and in the REPL.

There is one scenario, though, where is is not sufficient, and this is batch interpretation of a program, or even evaluation of the **start** function. When either of this happens, this is the order of initialization:

- the LISP9 system state is initialized
- an image is loaded
- batch (or startup) evaluation commences

However, loading an image will probably invalidate the cell reference stored in the **Rts** variable, so it no longer points to the stack. Of course, checks could be introduced to intercept this

case, but the simplest solution is to just store **Rts** in the image file.

The **Imagevars** variable is an array containing a list of image variables. Because it is a dynamically sized array, its last element must be **NULL** to indicate the end of the list.

```
cell    *Imagevars[] = {
        &Freelist, &Freevec, &Symbols, &Symhash,
        &Symptr, &Rts, &Glob, &Macros, &Obhash,
        &Obarray, &Obmap, NULL };
```

The **NULL**-terminated **GC_roots** array holds the addresses of internal variables whose values shall never be reclaimed by the garbage collector.

```
cell    *GC_roots[] = {
        &Protected, &Symbols, &Symhash, &Prog,
        &Env, &Obhash, &Obarray, &Obmap, &Cts,
        &Emitbuf, &Glob, &Macros, &Rts, &Acc,
        &E0, &Ep, &Argv, &Tmp, &Tmp_car,
        &Tmp_cdr, &Files, &Outstr, &Nullvec,
        &Nullstr, &Blank, &Zero, &One, &Ten,
        NULL };
```

20. The Command Line Interface

This chapter briefly describes the technicalities of starting a Unix command line program, like parsing the command line options. The most interesting part here is probably the proper order in which the individual parts of the LISP system are being initialized.

The **usage()**, **longusage()**, and **terms()** functions, respectively, print the following information when invoked:

- a brief summary of the command line options
- a long description of the command line options
- the terms of use (don't worry, LISP9 is in the public domain!)

```
void usage(void) {
    prints("Usage: ls9 [-Lhqv?] [-i file"
           " | -] [-l file]\n");
    prints("                [-- argument ..."
           " | file argument ...]\n");
}

void longusage(void) {
    nl();
    usage();
    prints("\n"
           "-h          "
           "print help (also -v, -?)\n"
           "-L          "
           "print terms of use\n"
           "-i file     "
           "restart image from file"
           " (default: ");
    prints(IMAGEFILE);
    prints(")\n"
           "-i -        "
           "compile initial image from"
           " sources (");
```

```
    prints(IMAGESRC);
    prints(")\n"
            "               "
            "(-i must be the first "
            "option!)\n");
    prints("-l file     "
            "load program from file, can be"
            " repeated\n"
            "-q         "
            "quiet (no banner, no prompt,"
            " exit on errors)\n"
            "-- args    "
            "bind remaining arguments to"
            " (cmdline)\n"
            "file args  "
            "run program, args in (cmdline),"
            "implies -q\n"
            "\n");
    exit(EXIT_SUCCESS);
}

void terms(void) {
    nl();
    prints( "LISP9 ");
    prints(VERSION);
    prints(" by Nils M Holm, 2018,2019\n\n"
    "This program is in the public domain. In countries\n"
    "where the concept of the public domain does not exist,"
    "\n"
    "the Creative Commons Zero (CC0) license applies.\n"
    "See: https://creativecommons.org/publicdomain/zero/1.0"
    );
    nl();
    nl();
    exit(EXIT_SUCCESS);
}
```

The **cmdarg()** function returns a command line argument or

signals an error when the given argument slot is unused (**NULL**). It is used to make sure that options requiring a value actually have one.

```
char *cmdarg(char *s) {
    if (NULL == s) {
        usage();
        exit(EXIT_FAILURE);
    }
    return s;
}
```

The **argvec()** function converts the argument vector **argv** to a list of strings and returns it. The resulting list will later be bound to the variable **Argv**.

```
cell    Argv = NIL;

cell argvec(char **argv) {
    int     i;
    cell    a, n;

    if (NULL == argv[0]) return NIL;
    a = cons(NIL, NIL);
    protect(a);
    for (i = 0; argv[i] != NULL; i++) {
        n = mkstr(argv[i], strlen(argv[i]));
        car(a) = n;
        if (argv[i+1] != NULL) {
            n = cons(NIL, NIL);
            cdr(a) = n;
            a = cdr(a);
        }
    }
    return unprot(1);
}
```

The **main()** function initializes the LISP system and then either interprets a single program in batch mode or enters the REPL. It also parses and interprets any command line options passed to the LISP system.

The fundamental startup sequence is as follows:

- The internal state is initialized via **init()**
- If an image file exists, it is loaded
- If no image file exists, the LISP9 library is loaded
- All files specified in **-l** options are loaded
- The **start** function is evaluated, if defined
- If a program name is specified, the program is interpreted and then the system will exit
- If no program is specified, the REPL is entered

The rest of the function deals with argument processing and building the **Argv** list.

One thing that might be interesting is the placement of various **setjmp()** calls throughout **main()**. Because error handling will transfer control to **Restart**, this context has to be established first and, depending on the position in **main()**, it will do different things:

(1) When **init()** or **loadimg()** should fail for any reason, it will exit silently in the hope that these functions will have printed some meaningful error message.

(2) When an error occurs while loading the LISP9 library source code, a fatal error will be signalled, indicating that the error occurred at this point of the initialization process.

(3) After loading the library code, the system will be reset to failing silently. Only when the REPL is entered, a different entry point for the error handler will be established. This means, in particular, that (uncaught) errors in batch mode will make the LISP system exit immediately.

```
int main(int argc, char **argv) {
```

```
int     i, j, k, usrimg, doload;
char    *s;
char    *imgfile;

imgfile = IMAGEFILE;
usrimg = 0;
doload = 1;
if (setjmp(Restart) != 0) exit(EXIT_FAILURE);
init();
i = 1;
if (argc > 2 && strcmp(argv[1], "-i") == 0) {
    imgfile = argv[2];
    i = 3;
    usrimg = 1;
}
if (existsp(imgfile) != NIL) {
    s = loadimg(imgfile);
    if (s != NULL) fatal(s);
    bindset(S_imagefile,
        mkstr(imgfile, strlen(imgfile)));
}
else if (usrimg && strcmp(imgfile, "-") != 0)
{
    fatal("cannot open image file");
}
else {
    if (setjmp(Restart) != 0)
        fatal("could not load library");
    loadfile(IMAGESRC);
}
if (setjmp(Restart) != 0) exit(EXIT_FAILURE);
for (; i<argc; i++) {
    if (argv[i][0] != '-') break;
    if ('-' == argv[i][1]) {
        doload = 0;
        break;
    }
    k = strlen(argv[i]);
```

```
        for (j=1; j<k; j++) {
            switch (argv[i][j]) {
            case '?':
            case 'h':
            case 'v':
                longusage();
                break;
            case 'L':
                terms();
                break;
            case 'l':
                i++;
                loadfile(cmdarg(argv[i]));
                j = strlen(argv[i]);
                break;
            case 'q':
                Quiet = 1;
                break;
            default:
                usage();
                exit(EXIT_FAILURE);
            }
        }
    }
    bindset(S_quiet, Quiet? TRUE: NIL);
    if (!Quiet && NULL == argv[i]) {
        prints("LISP9 "); prints(VERSION); nl();
    }
    Argv = NULL == argv[i]?
               NIL:
               argvec(&argv[i+1]);
    start();
    if (doload && argv[i] != NULL) {
        loadfile(argv[i]);
        exit(EXIT_SUCCESS);
    }
    repl();
    return 0;
```

```
}
```

PART II
LISP CODE

21. Derived Operations

```
;;; LISP9 Derived Syntax and Functions
;;; Nils M Holm, 2018,2019
;;; In the public domain
;;;
;;; If your country does not have a concept like
;;; the public domain, the Creative Common Zero
:;; (CC0) license applies, see
;;; https://creativecommons.org/publicdomain/zero/1.0
```

This part of the LISP system is referred to as its *library*, but it is not a library in the sense of a collection of functions which can be compiled into a program on demand. The LISP 9 library is a collection of interdependent functions and macros that will be compiled into the default *image file* of the system. The functions are always present and cannot be loaded separately. When the LISP9 system does not find an image file, it will load the code described in this part from its source file. When the source file is not present, the system cannot start up. All functions and derived special forms described here are a fixed part of the LISP9 language.

21.1 Simple Definitions

Reading the symbols **t** and **nil** at the beginning of this file will store them in *object table* slots 0 and 1. This is not strictly necessary, but facilitates debugging and may also enable certain optimizations, because the indexes 0 and 1 can then be used as truth values. None of these optimizations are performed in this implementation, though.

```
nil t
```

In the library code, a lot of definitions of the form

```
(defun (f x) (f x))
```

will appear. These definitions work only if *f* is the name of a *primitive function.* Since the LISP9 compiler will inline primitive functions, the application **(f x)** in the body of such a definition will generate inline code and not a function application. For instance, the definition

```
(defun (car x) (car x))
```

will inline a *car* instruction of the LISP9 abstract machine in the body of the **car** function. Hence applying the **car** function will actually perform a “car” operation. Definitions like these are necessary in order to compile expressions like

```
(mapcar car x)
```

where the name **car** does not appear in an operator position. Without redefining **car** as a function, the name **car** would be undefined in the above expression.

Most of these definitions are placed at the end of the library. Some of them are needed in the library code itself, though, and those are listed here.

```
(defun (cons x y) (cons x y))
(defun (car x) (car x))
(defun (cdr x) (cdr x))

(defun (caar x) (caar x))
(defun (cadr x) (cadr x))
(defun (cdar x) (cdar x))
(defun (cddr x) (cddr x))
```

Nested pair accessors with three levels of nesting (**caaar** ... **cdddr**) are also defined here, because they are used frequently in the following code. The single-level and two-level accessors are inlined by the compiler.

```
(defun (caaar x) (car (caar x)))
(defun (caadr x) (car (cadr x)))
```

```
(defun (cadar x) (car (cdar x)))
(defun (caddr x) (car (cddr x)))
(defun (cdaar x) (cdr (caar x)))
(defun (cdadr x) (cdr (cadr x)))
(defun (cddar x) (cdr (cdar x)))
(defun (cdddr x) (cdr (cddr x)))
```

The **list**, **string**, and **vector** functions create the corresponding type of object from a variable number or arguments. Using optional arguments, they are easy to implement.

```
(defun (list . x) x)
(defun (vector . x) (listvec x))
(defun (string . x) (liststr x))
```

The **rever** function reverses its argument, which must be a (proper) list. **Nrever** does the same, more efficiently, but at the cost of destroying its argument. They are easily implemented on top of **reconc** and **nreconc**.

```
(defun (rever a)
  (reconc a nil))

(defun (nrever a)
  (nreconc a nil))
```

21.2 Bootstrapping

A lot of definitions in the LISP9 library depend on each other, which creates the classical problem of *bootstrapping*: definition A depends on definition B and B depends on A. In the case of derived special forms, there is an easy (but cumbersome) way around: use a manually expanded form of B in A. The unwieldiness of manually expanded definitions can be mitigated by introducing definitions gradually. For instance, even a simplified **let** or **cond** special form can be very helpful, and the full implementation can be postponed until more expressive tools are

available.

The same approach can be used with functions. For instance, a simplified **mapcar** function will be defined early in the bootstrapping process. It is sufficient for all tasks to be solved in the library itself, and the full implementation only appears rather late in the process.

As a first step in the bootstrapping process, a simplified form of **cond** is defined. This variant accepts only clauses of the form

```
(predicate expression ...)
```

and expands as follows:

```
(cond (p1  x1,1 ···)     →   (if p1
      ···                         (prog x1,1 ···)
      (pn  xn,1 ···))             (if ···
                                    ···
                                      (if pn
                                          (prog xn,1 ···)
                                          nil) ··· ))
```

The predicate of a clause may be the keyword **else**, which is always true. Such a clause expands in this way:

```
(else x1 ...)   →   (prog x1 ...)
```

The definition of a minimal **cond** special form is straightforward, but the expanded form has to be constructed by using **list** and **cons**, because quasiquotation is not yet available at this point. Once it is, a complete **cond** macro will be defined.

```
(defmac (cond . cs)
  (if (null cs)
      nil
      (if (eq 'else (caar cs))
          (cons 'prog (cdar cs))
          (list 'if (caar cs)
                    (cons 'prog (cdar cs))
                    (cons 'cond (cdr cs))))))
```

The **and** special form implements the *short-circuit logical and*, i.e. it returns the first of its arguments that is **nil** or the last argument when no argument is **nil** (and **t**, if no arguments are given). It implements a *short-circuit* operator, because it will not evaluate any further arguments as soon as the result can no longer change: when one argument is **nil**, it does not make any sense to evaluate anything that follows.

The **and** special form expands in the following way:

$$\texttt{(and } x_1 \ x_2 \cdots x_n\texttt{)} \quad \rightarrow \quad \texttt{(if } x_1 \ \texttt{(and } x_2 \cdots x_n\texttt{)} \ \texttt{nil)}$$

Multiple arguments expand recursively.

```
(defmac (and . xs)
  (cond ((null xs) 't)
        ((null (cdr xs)) (car xs))
        (else (list 'if (car xs)
                        (cons 'and (cdr xs))
                       nil))))
```

Quasiquotation is a convenient tool in the transformation of derived special forms. It is similar to quotation, but allows to insert dynamic elements into list and vector structures. The structure being processed by quasiquotation is called a *quasiquotation template*.

Quasiquotation consists of the **qquote** special form and the **unquote** and **splice** subforms, which are only valid inside of **qquote**. The following abbreviations are commonly used:

```
 @x  =  (qquote x)
 ,x  =  (unquote)
,@x  =  (splice x)
```

Note that Scheme and Common Lisp would use the backtick (‘) character in the place of “@”, but not in the place of “,@”. E.g., it would use **‘(f ,@x)** in the place of **@(f ,@x)**. The backtick character can also be used in LISP9.

Given the above abbreviations, this is how quasiquotation works formally:

```
      @x   =   'x
     @,x   =   x
  @(x y)   =   (cons @x @(y))
 @(,x y)   =   (cons x @(y))
@(,@x y)   =   (conc x @(y))
```

Note particularly that there is a **conc** in the last equation, but a **cons** in the other formulae. The difference might not be obvious, so here are some examples:

```
@(1 ,(list 2 3) 4)
```

would expand to

```
(cons '1 (cons (list 2 3) (cons '4 'nil)))
  =>  (1 (2 3) 4)
```

but

```
@(1 ,@(list 2 3) 4)
```

would expand to

```
(cons '1 (CONC (list 2 3) (cons '4 'nil)))
  =>  (1 2 3 4)
```

That is, the **splice** operator *splices* multiples values into the place of its application instead of inserting a list of values (which is what the **unquote** operator would do).

Because **qquote** expands recursively, it can be used to insert elements into templates of any complexity. For instance, given a list of variables $vs = (x\ \ y\ \ z)$ and a list of arguments $as = (A\ \ B\ \ C)$ and a list of expressions $body = (x_1\ \ x_2\ \ x_3)$, **qquote** can be used to create an S-expression denoting function application:

```
@((lambda ,vs ,@body) ,@as)
  =>  ((lambda (x y z) x1 x2 x3) A B C)
```

Note that splicing is not really necessary at the end of a list, so the following template would also work:

```
@((lambda ,vs . ,body) . ,as)
```

Splicing only has to be used when multiple values are to be inserted into a list where further list elements follow after the place where the insertion takes place. E.g.:

```
@(a b . (unquote '(0 1)))   =>   (a b 0 1)
@(a (splice '(0 1)) b)      =>   (a 0 1 b)
@((splice '(0 1)) a b)      =>   (0 1 a b)
```

See the definition of **labels** for a more practical example.

The **qquote** special form itself is implemented as an ordinary macro. Only support for the abbreviations listed above has to be provided by the reader.

The code is straight-forward, it basically implements the formal rules outlined earlier in this description. A special case that was not discussed above, though, is the use of a vector in a template. This case is handled by converting the vector to a list, expanding the list, and then converting it back to a vector.

```
(defmac (qquote x)
  (cond ((vectorp x)
          (list 'listvec
                (list 'qquote (veclist x))))
        ((not (pair x))
          (list 'quote x))
        ((eq 'unquote (car x))
          (cadr x))
        ((and (pair (car x))
              (eq 'unquote (caar x)))
          (list 'cons (cadar x)
                      (list 'qquote (cdr x))))
        ((and (pair (car x))
              (eq 'splice (caar x)))
          (list 'conc (cadar x)
                      (list 'qquote (cdr x))))
        (else
          (list 'cons (list 'qquote (car x))
                      (list 'qquote (cdr x))))))
```

21.2.1 Binding Constructs

Note that the name *binding* will be used for two related things in this part of the text. The first meaning is known from earlier chapters and refers to the association of a variable name to a storage location. The other thing that will be called a "binding" in this chapter is an element of that part of the **let**, **labels**, and **do** forms that has the shape

```
((variable expression) ···)
```

This use of the word is, of course, closely related to the first meaning, because what forms like **let** do *is* to create bindings in the original sense.

The **let** special form performs the transformation

```
(let ((v1 a1)   →   ((lambda (v1 ··· vn)
       ···                  x ···)
      (vn an))           a1 ··· an)
  x ···)
```

This transformation would be easy to implement, if local functions could be defined using **labels**, but **labels** is not yet available, so some improvisation is called for.

The **let** macro defines a local function named *split*, which will be used to split bindings of the form **(v a)** into variables (v) and arguments (a).

Because **labels** cannot be used, **let** creates a lambda function with a formal argument named *split* and applies it to **nil**, thereby creating a local variable named *split*. In the context where this local variable exists, the "split" function can then be assigned to that variable using **setq**. Because the variable is free in the *body* of the outer function of *split*, *split* can recurse. See page 156 or the definition of **labels**, further below, for a detailed explanation. In fact, the code of **let** uses a manually macro-expanded **labels** form.

After splitting the bindings, the variables are bound to *vs* and the arguments to *as* and then a quasiquotation template is used to construct an equivalent lambda special form.

```
(defmac (let bs x . xs)
  ((lambda (split)
     (setq split
           (lambda (bs vs as)
             (if (null bs)
                 (list vs as)
                 (split (cdr bs)
                        (cons (caar bs) vs)
                        (cons (cadar bs) as)))))
     (apply (lambda (vs as)
              @((lambda ,vs ,x . ,xs) . ,as))
            (split bs nil nil)))
   nil))
```

The **mapcar** function is a higher-order function that maps a function over some given lists. A *higher order function* is a "function of functions", i.e. a function that either accepts another function as its argument or returns a function or both.

Mapping a function f over a list a means to create a new list that contains f applied to each element of a:

(map f a**)** = **(list (**f a_1**)** $\cdots$ **(**f a_n**))**

or, in case it is applied to two lists:

(map f a b**)** = **(list (**f a_1 b_1**)** $\cdots$ **(**f a_n b_n**))**

The full **mapcar** function accepts any (positive) number of list arguments, but it is too hard to implement without using **labels**, so a minimal version is defined here that only accepts one or two list arguments. In fact this version contains two separate local functions handling the individual cases: **m1** maps f over a and **m2** maps f over a and b. The local functions are defined by using a macro-expanded **labels** special form, just like in **let**, above.

```
(defun (mapcar f a . b)
  (if (null b)
      (let ((m1 nil))
        (setq m1 (lambda (a)
```

```
                    (if (null a)
                        nil
                        (cons (f (car a))
                              (m1 (cdr a))))))
          (m1 a))
        (let ((m2 nil))
          (setq m2 (lambda (a b)
                    (if (null a)
                        nil
                        (cons (f (car a)
                                 (car b))
                              (m2 (cdr a)
                                  (cdr b))))))
          (m2 a (car b)))))
```

The **labels** special form is similar to **let**, but its purpose is to define *local functions*, i.e. functions that are defined inside of the body of another function (although **labels** can also be used as a top-level expression).

It performs the transformation

```
(labels ((v1 a1)      →    (let ((v1 nil)
         ...                      ...
         (vn an))                 (vn nil))
  x1 ···)                    (setq v1 a1)
                             ...
                             (setq vn an)
                             x1 ···)
```

The whole point of the transformation is to make the variables $v_1 \cdots v_n$ *free* in the context where the expressions $a_1 \cdots a_n$ are evaluated. This is done by binding them to **nil** in the outer **let** special form. Because the variables are free, a name in **setq** and a name referenced in an expression $a_i = (lambda \cdots)$ would refer to *the same variable*. Hence each name v_i can be used in each a_i to refer to any other a_i.

To illustrate this point, the assumption that the same effect can be achieved by just using **let** is proven wrong. The form

```
(let ((f (lambda () (f)))) (f))
```

expands to

```
((lambda (f) (f)) (lambda () (f)))
```

Here the right lambda closes over the name f, creating a free variable f_1 with an undefined value. Then the resulting function is passed to the left lambda function, which binds it to its argument f_2, which is different from f_1. When f_2 is applied to no arguments in the left function, the right function will in turn apply f_1 to no arguments, but f_1 is still undefined. In other words, the above expression is not different from

```
((lambda (f) (f)) (lambda () (undefined)))
```

The special form

```
(labels ((f (lambda () (f)))) (f))
```

expands to

```
((lambda (f)
   (setq f (lambda () (f)))
   (f))
 nil)
```

Here the variable of **setq**, the variable referenced in **(lambda () (f))**, and the variable in **(f)** are the same free variable, which is only bound in the outermost **lambda**. The variable is bound in the **lambda**, but free in its body.

Note that the **labels** special form described here differs from Scheme's **letrec** [R4RS], because it allows expressions a_i to refer to variables defined previously in the same special form. It is a cross between **letrec** and **let***, which is in some Scheme dialects known as **letrec***. For instance, the following expression works in LISP9, but would not work when using Scheme's **letrec** instead of **labels**:

```
(labels ((compose
           (lambda (f g)
             (lambda x
               (f (apply g x)))))
         (abs- (compose abs -)))
```

```
  (abs- 7 5))                                  =>   2
```

The use of **mapcar** simplifies the construction of **labels** a lot. The first two applications of **mapcar** extract the variables and expressions from the bindings of the special form and store them in *vs* and *as*, respectively. Then a list of bindings for **let** of the form **(v nil)** is created and stored in *ns* ("NILs") and a list of expressions of the form **(setq v a)** is stored in *is* ("inits"). In each of these lists the *v*'s and *a*'s are replaced with the variable names and expressions passed to **labels**.

Finally, the expanded form is created using **qquote**.

```
(defmac (labels bs x . xs)
  (let ((vs (mapcar car bs))
        (as (mapcar cadr bs)))
    (let ((ns (mapcar (lambda (v)
                        (list v nil))
                      vs))
          (is (mapcar (lambda (v a)
                        (list 'setq v a))
                      vs
                      as)))
      @(let ,ns ,@is ,x . ,xs))))
```

21.3 Conditional Evaluation

The **or** special form is similar to **and**: it implements the short-circuit *logical or*. The form expands as follows:

$$\texttt{(or } x_1\ x_2\ \cdots\ x_n\texttt{)} \quad \rightarrow \quad \texttt{(if* } x_1\ \texttt{(or } x_2\ \cdots\ x_n\texttt{))}$$

The definition of the **or** form is one place where the **if*** special form comes in handy. The other one is single-expression clauses in the full **cond**, as defined below.

```
(defmac (or . xs)
  (cond ((null xs) nil)
        ((null (cdr xs)) (car xs))
```

```
          (else @(if* ,(car xs)
                      (or . ,(cdr xs))))))
```

With the help of the minimal **cond** and quasiquotation, a full implementation of the **cond** special form can now be provided. It works exactly like the minimal variant, but performs the following transformations in addition.

(1) A clause that contains only a predicate *p* will be rewritten as follows:

```
(cond (p) . cs)  →  (if* p (cond . ,(cdr cs)))
```

where *cs* denotes any subsequent clauses of **cond**.

(2) A clause of the form

```
(p => f)
```

where *p* is a predicate and *f* is a function of one argument, will be rewritten as follows:

```
(cond (p => f) . cs)
                   →  (let ((G p))
                         (if G
                             (f G)
                             (cond . ,(cdr cs))))
```

where the symbol *G* will be generated by **gensym**. The effect of such a clause is that *f* will be applied to the result of *p*, if that result is not **nil**, and *p* will only be evaluated once in either case. For instance:

```
(cond ((assoc 'a '((a 1))) => cdr))  =>  (1)
```

Note that the left **=>** is part of the **cond** syntax in this example, but the right one is the meta-syntactic "evaluates to" operator.

The implementation of this kind of clause in particular is simplified greatly by the use of quasiquotation.

```
(defmac (cond . cs)
  (cond ((null cs) nil)
        ((null (cdar cs))
```

```
          @(if* ,(caar cs)
                (cond . ,(cdr cs))))
        ((eq '=> (cadar cs))
          (let ((g (gensym)))
            @(let ((,g ,(caar cs)))
                (if ,g (,(caddr (car cs)) ,g)
                    (cond . ,(cdr cs))))))
        ((eq 'else (caar cs))
          @(prog . ,(cdar cs)))
        ((null (cdr cs))
          @(if ,(caar cs)
                (prog . ,(cdar cs))))
        (else
          @(if ,(caar cs)
                (prog . ,(cdar cs))
                (cond . ,(cdr cs))))))
```

The **case** special form is the same as in Scheme [R4RS]. It expands as follows, where each L_i is a flat list of objects and k ("key") is any expression:

```
(case k              →   (let ((G k))   ; G = (gensym)
  (L1 x1,1 ···)            (if (memv G L1)
  ···                          (prog x1,1 ···)
  (Ln xn,1 ···))               (if ···
                                  ···
                                  (if (memv G Ln)
                                      (prog xn,1 ···)
                                      nil)···)))
```

In addition, a clause with the keyword **else** in the place of a list will expand as follows:

```
(else x ···)   =>   (prog x ···)
```

Informally, the **case** special form evaluates the expressions $x \cdots$ of the first clause whose list part L contains the value of k. Membership of k in L is tested via the **memv** function, so both, the value of k and objects in L, must be comparable using **eqv** (q.v.).

```
(defmac (case x . cs)
  (defun (cases x cs)
    (cond ((null cs) nil)
          ((eq 'else (caar cs))
            @(prog . ,(cdar cs)))
          (else
            @(if (memv ,x ',(caar cs))
                 (prog . ,(cdar cs))
                 ,(cases x (cdr cs))))))
  (let ((g (gensym)))
    @(let ((,g ,x))
       ,(cases g cs))))
```

21.4 Binding Forms

The **let*** special form is like **let**, but binds their variables *sequentially* while **let** binds them in no specific order. Hence **let** is more efficient, but **let*** is more predictable. The **let*** special form is transformed to a nested application of single-variable **let**:

```
(let* ((v1 a1)        →   (let ((v1 a1))
       ...                  ...
       (vn an))               (let ((vn an))
  x ···)                        (let () x ···))···)
```

Note that a **let*** special form with no bindings evaluates to a **let** special form with no bindings. This is done because

```
(let () x)
```

transforms to

```
((lambda () x))
```

which in turn closes over any free variables in x. **Let*** should not behave differently in this case.

```
(defmac (let* bs x . xs)
  (if (null bs)
      @(let () ,x . ,xs)
```

```
        @(let (,(car bs))
           (let* ,(cdr bs) ,x . ,xs))))
```

LISP9 also supports what Scheme [R4RS] calls *named let* and which actually is a very general from of iteration, comparable to the unbounded "while" loop in other programming languages.

The following **let** macro extends the **let** macro defined earlier in this part of the book. Note that this version uses the previous version of **let** internally, which is not a problem, because macro expansion takes place before the new macro definition is registered in the macro environment. (BTW, this has already been done with **cond**, earlier in this text, in case you wondered!)

The new **let** has an additional argument, $x0$, which is either a list of bindings or a symbol. When it is a list of bindings, expansion works in the same way as for the simplified **let**. When $x0$ is a symbol, though, the special form expands as follows:

```
(let name          →   ((labels
     ((v1 a1)              ((name
       ···                    (lambda (v1 ··· vn)
      (v1 an))                  x ···)))
  x ···)                   name)
                          a1 ··· an)
```

The **labels** form in the resulting code creates a recursive function named *name* and returns it. The function *name* has the variables v_1 through v_n as its formal arguments. The entire **labels** form is then applied to the arguments a_1 through a_n, thereby applying *name* to their values. Because *name* is recursive, the loop can now be reentered by applying the function to a new set of values.

Because such a loop can iterate until an arbitrary condition is met, it implements *general recursion*, i.e. the form of recursion that can compute any computable function. See, for instance [Smith1996].

```
(defmac (let x0 x . xs)
  (if (symbolp x0)
      (let ((vs (mapcar car x))
            (as (mapcar cadr x)))
```

```
        @((labels ((,x0 (lambda ,vs . ,xs)))
                  ,x0) . ,as))
      (let ((vs (mapcar car x0))
            (as (mapcar cadr x0)))
        @((lambda ,vs ,x . ,xs) . ,as))))
```

The **with** special form is, strictly speaking, not a *binding* form at all, because it temporarily alters the values stored in the locations bound to *free* variables. Therefore, all variables listed in the "bindings" part of the **with** form must already exist.

The purpose of the **with** special form is to alter the values of free variables in its *dynamic extent.* The dynamic extent of a form is a temporal rather than spatial phenomenon. It starts when a form is "entered", i.e. when the flow of control passes into the form, and extends up to the point where the flow of control returns from the form.

For example, the program

```
(def *p* 'outer-value)
(let ((p *p*))
  (setq *p* 'inner-value)
  (F)
  (setq *p* p))
```

has the effect of changing the value of the free variable $*p*$ temporarily. The change remains in effect while the function F evaluates. Therefore, the dynamic extent of the effect is equal to the body of the function F.

Note that the location of the function F does not matter. As long as it shares the free variable $*p*$ with the above program, the effect will be visible in F.

Other functions with an effect that is visible in their dynamic extent include **catch*** and **catch**, where a catch tag remains valid in their dynamic extent, or **with-infile** and **with-outfile**, which temporarily reconnect the current input and output port inside of their dynamic extent.

The effect of the **with** special form is the same as that of the

above example program. Syntactically, **with** looks like **let**, but it expands in a completely different way:

```
(with ((v1 a1)      →   ((lambda (G1 ··· Gn)
       ···                  (unwind
       (vn an))               (lambda ()
  x ···)                        (setq v1 G1)
                                 ···
                                (setq vn Gn)))
                              (lambda ()
                                (setq v1 a1)
                                 ···
                                (setq vn an)
                                x ···)))
                          v1 ··· vn)
```

The **unwind** function will be explained later. At this point it shall suffice to know that **(unwind** g f**)** first evaluates f, then evaluates g, and eventually returns the value returned by f.

The outer lambda function of the expanded form of **with** binds the values of the given free variables v_i to the temporary locations G_i, where the names G_i are generated by **gensym**. It then alters the value of each v_i and with those altered values in effect it evaluates the body $x \cdots$ of the form. Before returning its value, it also evaluates the first lambda function in **unwind**, which restores the previous value of each v_i.

Here is a small example using **with:**

```
(def *name* 'unknown)

(defun (hello) (list 'hello *name*))

(hello)                          =>   (hello unknown)

(with ((*name* 'world))
  (hello))                       =>   (hello world)
```

That is, the **with** special form implements what is sometimes known as “*dynamic scoping*”. In some dialects of LISP it is known as **fluid-let**.

By convention, variables that are expected to be altered by **with**,

fluid-let, or comparable special forms start and begin with an asterisk character to indicate that their values may change dynamically due to effects that originate outside of the current lexical (spatial) scope. This convention is widely known as the "earmuff" convention, because variables marked in this way look like they were wearing earmuffs. The variable **foo** may be pronounced "earmuff-foo" or "foo-earmuff".

```
(defmac (with bs x . xs)
  (let* ((vs  (mapcar car bs))
         (gs  (mapcar (lambda (x) (gensym))
                      bs))
         (as  (mapcar cadr bs))
         (set (mapcar (lambda (v a)
                        @(setq ,v ,a))
                      vs
                      as))
         (res (mapcar (lambda (v g)
                        @(setq ,v ,g))
                      vs
                      gs))
         (val (gensym)))
    @((lambda ,gs
        (unwind
          (lambda () . ,res)
          (lambda ()
            ,@set
            ,x . ,xs))) . ,vs)))
```

The **do** special form is another general recursive looping construct. Its syntax is more terse than named **let**, but it has only one exit condition and cannot pass control freely between nested instances of itself (which named **let** can do, because the loops are named).

The **do** form is a true binding form, because it creates bindings, just like **let** or **labels**. However, the bindings of **do** have an additional – and optional – part called the *increment* or *step*.

When no increment it specified in a binding, it defaults to the variable. E.g. the binding **(v a)** would be equal to **(v a v)**.

The variable (v) and argument (a) parts of a **do** binding serve the same purpose as in the other binding forms. The increment/step (s) part is what is passed to the looping function in subsequent iterations. For instance, the following expression would create a list of the number from 0 through 99.

```
(do ((n 0 (+ 1 n))
     (a nil (cons n a)))
    ((= n 100) (rever a)))
```

The second argument of **do** is where the test for termination takes place. It is basically like a **cond** clause: when the first member of the clause (the *predicate*) is true, the other members are evaluated as an implied **prog** form. Otherwise, when the predicate is false, the *body* of the form is evaluated and the loop starts over. In this case the increment parts of the bindings provide the values for the bound variables in the next iteration. The body of the loop is empty in the above example.

The **do** special form expands as follows:

```
(do ((v1 a1 s1)        →   (labels
     ...                     ((G (lambda (v1 ... vn)
     (vn an sn))                  (if test
    (test r ...)                      (prog r ...)
 x ...))                              (prog
                                        x ...
                                        (G s1 ...
                                           sn)))))
                             (G a1 ... an))
```

As in the previous equations, G represents a symbol created by **gensym**.

The case where the increment parts are missing from the bindings is omitted in the above expansion. For each binding without an increment part, the corresponding s_i in the expanded form it replaced with v_i.

```
(defmac (do bs tst . xs)
  (let ((fn (gensym))
        (vs (mapcar car bs))
        (as (mapcar cadr bs))
        (ss (mapcar cddr bs)))
    (let ((ss (mapcar (lambda (s v)
                        (if (null s) v (car s)))
                      ss vs)))
      @(labels
         ((,fn (lambda ,vs
                 (if ,(car tst)
                     (prog . ,(cdr tst))
                     (prog ,@xs (,fn . ,ss))))))
         (,fn . ,as)))))
```

21.5 Higher-Order Functions

This section will define some more *higher-order functions* of the LISP9 library, among others a full (variadic) implementation of **mapcar**.

The **fold** function *folds* the function f over the elements of the list a, where the variable b serves as a *base element* or *neutral element*. **Foldr** is like **fold**, but folds the argument a to the right. Formally,

$$\texttt{(fold } f\ b\ \texttt{'(}a_1\ \cdots\ a_n\texttt{))} = \texttt{(}f\ \cdots\ \texttt{(}f\ \texttt{(}f\ b\ a_1\texttt{)}\ a_2\texttt{)}\ \ldots\ a_n\texttt{)}$$

and

$$\texttt{(foldr } f\ b\ \texttt{'(}a_1\ \cdots\ a_n\texttt{))} = \texttt{(}f\ a_1\ \texttt{(}f\ a_2\ \cdots\ \texttt{(}f\ a_n\ b\texttt{)}\ \cdots\ \texttt{))}$$

So, for example:

```
(fold  cons nil '(a b c))  =>  (((nil . a) . b) . c)
(foldr cons nil '(a b c))  =>  (a . (b . (c . nil)))
```

Both **fold** and **foldr** use an internal function that collects the result in an accumulator (r, “result”) in order to evaluate in constant space.

```
(defun (fold f b a)
  (defun (fl a r)
    (if (null a)
        r
        (fl (cdr a)
            (f r (car a)))))
  (fl a b))

(defun (foldr f b a)
  (defun (fr a r)
    (if (null a)
        r
        (fr (cdr a)
            (f (car a) r))))
  (fr (rever a) b))
```

The **filter** function creates a fresh list that contains only elements of its argument a that satisfy the predicate p. Formally,

(filter *p a***)**

equals

```
(apply conc
       (map (lambda (x)
              (if (p x) (list x) nil))
            a))
```

but the actual implementation avoids the boxing of the list elements and the final application of **conc** and is therefore much more efficient.

```
(defun (filter p a)
  (defun (fi a r)
    (cond ((null a) (nrever r))
          ((p (car a))
            (fi (cdr a) (cons (car a) r)))
          (else
            (fi (cdr a) r))))
```

```
  (fi a nil))
```

The **memq** function, which finds the first tail of a that starts with x, is required to implement **mapcar**, below. It is a standard Scheme function [R4RS] and also part of the LISP9 library.

```
(defun (memq x a)
  (cond ((null a) nil)
        ((eq x (car a)) a)
        (else (memq x (cdr a)))))
```

The following implementation of the **mapcar** function is a generalization of the **mapcar** function implemented at the beginning of this chapter. It accepts any (positive) number of list arguments. In principle it is very similar to the following single-list implementation:

```
(defun (mapcar f a)
  (defun (map2 a r)
    (if (null as)
        (nrever r)
        (map2 (cdr a)
              (cons (f (car a))
                    r))))
  (map2 a nil))
```

However, it uses **car*** in the place of **car**, **cdr*** in the place of **cdr**, **nil*** in the place of **null**, and uses **apply** to apply f to a variable number of arguments. The modified versions of **car**, etc do not operate on lists, but on lists of lists:

```
(car* '((1 2) (3 4) (5 6)))  =>  (1 3 5)
(cdr* '((1 2) (3 4) (5 6)))  =>  ((2) (4) (6))
```

Hence **car*** extracts the arguments for an application of f and **cdr*** advances to the next set of elements, just like **car** and **cdr** in the single-argument version. **Nil*** returns truth, if any list in its list-of-lists argument is **nil**:

```
(nil* '((1) (2) (3) (4)))  =>  nil
```

```
(nil* '((1) (2) nil (4)))  =>  t
```

Finally, **mapcar** has to implement its own single-list mini-**mapcar** (named **map**) in order to avoid infinite recursion through **car*** or **cdr***.

```
(defun (mapcar f a . as)
  (defun (map f x)
    (if (null x)
        nil
        (cons (f (car x))
              (map f (cdr x)))))
  (defun (car* x) (map car x))
  (defun (cdr* x) (map cdr x))
  (defun (nil* x) (memq nil x))
  (defun (mapcar* as r)
    (if (nil* as)
        (nrever r)
        (mapcar* (cdr* as)
                 (cons (apply f (car* as))
                       r))))
  (mapcar* (cons a as) nil))
```

The **foreach** function is like **mapcar**, but is guaranteed to apply f to the elements of a from the left to the right. It also returns an uninteresting result. Because **mapcar** already does process its arguments in the desired order, **foreach** can delegate its work to it.

```
(defun (foreach f a . as)
  (apply mapcar f a as)
  nil)
```

21.6 List Functions

The **length** function returns the length of a proper list and signals an error when a dotted list is passed to it.

```
(defun (length ls)
  (defun (len a n)
    (cond ((null a) n)
          ((pair a) (len (cdr a) (+ 1 n)))
          (else (error
                  "length: improper list"
                  ls))))
  (len ls 0))
```

The **nth** function returns the nth element of a list and the **nth-tail** function returns the tail of a list that starts at the nth element. A tail returned by **nth-tail** may be **nil**.

```
(defun (nth-tail n a)
  (if (= 0 n)
      a
      (nth-tail (- n 1) (cdr a))))

(defun (nth n a) (car (nth-tail n a)))
```

The **listp** predicate returns truth, if its argument x is a *proper list*, i.e. if it is *finite* structure of conses that ends with **nil**. The case where x is a dotted list is easy to catch by traversing the list and examining its last cdr part. The case where x is not a cons cell at all is even easier.

However, when x is *cyclic* cons structure. the naive algorithm would run into an infinite loop, because there is no "last" cdr part to check. E.g.:

```
(naive-listp (let ((x (list 1))) (setcdr x x)))
```

would never terminate.

A common algorithm to detect cycles in a list is Robert W. Floyd's *tortoise and hare* algorithm [Knuth1969], where a "fast" pointer traversing the list chases a "slow" pointer. The fast pointer always skips one element while the slow pointer traverses each single

element. If the pointers should ever point to the same element, the traversed structure is cyclic. The algorithm is guaranteed to terminate *and* to find any cycle in a list.

The proof of termination can be found in figure 53. Each cycle has either an even or an odd number of elements. When the cycle has two elements, there are two cases to cover: (1) the fast pointer F and the slow pointer S point to the same cell. In this case, the algorithm terminates immediately. (2) S and F point to different cells. In this case F advances to the same cell in the next step and S will advance to the other cell, leading to $S = F$, and hence termination. This is illustrated in the top row of the figure.

When there are three cells in the cycle, there are three cases to cover, which are depicted in the bottom row of the figure: (1) $S = F$ trivially terminates. (2) S precedes F. In this case $S = F$ will be reached after two steps. (3) F precedes S. In this case $S = F$ will be reached in the next step. This is shown in the bottom row of the figure. The general cases are proved by induction.

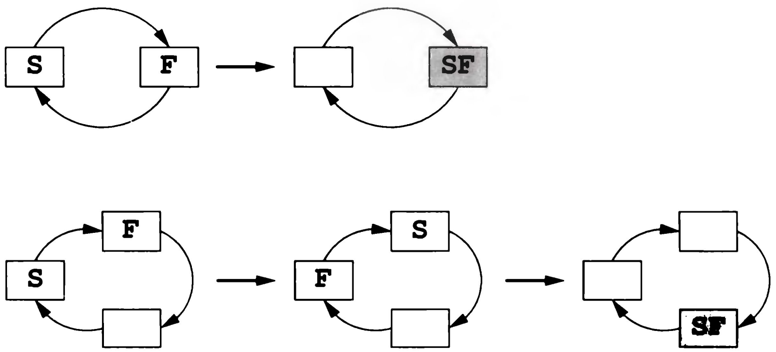


Fig. 53 – Tortoise and hare algorithm

The algorithm is guaranteed to find any cycle in a list, because if there is a cycle in a list, traversing the list *must* enter the cycle.

The *acyclicp* function that is local to **listp** will return truth when the fast pointer x and the slow pointer y never reach the same element. Of course, a function applying *acyclicp* must make sure that x and y point to different elements initially (if that is possible at all).

```
(defun (listp x)
  (defun (acyclicp x y)
     (cond ((eq x y) nil)
           ((null x))
           ((pair x)
             (or (null (cdr x))
                 (and (pair (cdr x))
                      (acyclicp (cddr x)
                                (cdr y)))))
           (else nil)))
  (or (null x)
      (and (pair x)
           (acyclicp (cdr x) x))))
```

Nested pair accessors with four levels of nesting (**caaaar** ... **cddddr**) are defined here. The two-level accessore are inlined.

```
(defun (caaaar x) (caar (caar x)))
(defun (caaadr x) (caar (cadr x)))
(defun (caadar x) (caar (cdar x)))
(defun (caaddr x) (caar (cddr x)))
(defun (cadaar x) (cadr (caar x)))
(defun (cadadr x) (cadr (cadr x)))
(defun (caddar x) (cadr (cdar x)))
(defun (cadddr x) (cadr (cddr x)))
(defun (cdaaar x) (cdar (caar x)))
(defun (cdaadr x) (cdar (cadr x)))
(defun (cdadar x) (cdar (cdar x)))
(defun (cdaddr x) (cdar (cddr x)))
(defun (cddaar x) (cddr (caar x)))
(defun (cddadr x) (cddr (cadr x)))
(defun (cdddar x) (cddr (cdar x)))
(defun (cddddr x) (cddr (cddr x)))
```

21.7 Equivalence

The **eqv** predicate tests whether its arguments are *equivalent*.

Equivalence is a weaker relation than identity (or sameness), but a stronger relation than structural equality.

Two objects are *identical* if and only if they are *the same object*, i.e. if they occupy the same slot in the node pool or they are the same special object. Hence a test for identity can be performed by using C's == operator.

When returned by the reader, only symbols and special objects can ever be identical in LISP9. For all other atoms, the identity operation, implemented by the **eq** function, is undefined. For example:

```
(eq 'foo 'foo)   =>  t
(eq nil  nil)    =>  t
(eq 123  123)    =>  undefined
```

However,

```
(let ((x (list 1))) (eq x x))  =>  t
```

because the object stored in the binding of x is, of course, identical to itself.

Equivalence covers and extends the concept of identity. Two objects a and b are equivalent, if

- they are identical
- they are both fixnums and **(=** a b**)** is true
- they are both chars and **(c=** a b**)** is true

The equivalence relation is undefined on strings, vectors, and lists, except when objects of those types are identical.

```
(defun (eqv a b)
  (cond ((eq a b))
        ((and (fixp a)
              (fixp b)
              (= a b)))
        ((and (charp a)
              (charp b)
              (c= a b)))
        (else nil)))
```

The **equal** predicate returns truth, if its arguments are structurally equal. Two objects a and b are structurally equal, if

- they are equivalent
- they are both strings of the same length and **(c=** a_i b_i**)** for $0 \leq i < |a|$ (where a_i is the ith char of a)
- they are both vectors and **(equal (veclist** a**) (veclist** b**))** is true
- they are both lists and **(equal (car** a**) (car** b**))** and **(equal (cdr** a**) (cdr** b**))** are both true

Informally, two objects are structurally equal, if the **prin** function prints them in the same way. This is not completely true, though, because some objects that do not have an unambiguous external representation will print the same but still may be different. For instance,

```
(equal (lambda (x) x)
       (lambda (x) x))  =>  undefined
```

Equal takes a shortcut by testing for identity first, which is a cheap operation. Doing so avoids the element-by-element comparison of larger structures that are identical (and hence equal) anyway.

Also note that vectors are compared element by element by the internal *equvec* function instead of converting them to lists first. This approach saves the creation of $n + m - k$ nodes: n conses to convert list a plus m conses to convert list b minus a little storage (k) that is allocated for the local variable i.

```
(defun (equal a b)
  (defun (equvec a b)
    (and (= (vsize a) (vsize b))
         (let loop ((i (- (vsize a) 1)))
           (cond ((< i 0))
                 ((equal (vref a i) (vref b i))
                   (loop (- i 1)))
                 (else nil)))))
  (cond ((eq a b))
```

```
        ((and (pair a)
              (pair b)
              (equal (car a) (car b))
              (equal (cdr a) (cdr b))))
        ((and (stringp a)
              (stringp b)
              (s= a b)))
        ((and (vectorp a)
              (vectorp b)
              (equvec a b)))
        (else (eqv a b))))
```

Memv is like **memq**, but uses **eqv** in the place of **eq** to compare elements. **Member** is also like **memq**, but uses **equal** to compare elements.

```
(defun (memv x a)
  (cond ((null a) nil)
        ((eqv x (car a)) a)
        (else (memv x (cdr a)))))

(defun (member x a)
  (cond ((null a) nil)
        ((equal x (car a)) a)
        (else (member x (cdr a)))))
```

The **assq** function is the LISP version of the C function **assq()**, which has been defined earlier in this text (see page 90). **Assv** and **assoc** are like **assq**, but use **eqv** and **equal** instead of **eq** to compare keys.

```
(defun (assq x a)
  (cond ((null a) nil)
        ((eq x (caar a)) (car a))
        (else (assq x (cdr a)))))

(defun (assv x a)
```

```
  (cond ((null a) nil)
        ((eqv x (caar a)) (car a))
        (else (assv x (cdr a)))))

(defun (assoc x a)
  (cond ((null a) nil)
        ((equal x (caar a)) (car a))
        (else (assoc x (cdr a)))))
```

21.8 Arithmetic Functions

The following macros are shortcuts for the more common bitwise logical operations (see **bitop**, page 244). The function definitions that follow after the macros are the **apply**-able versions of these operators, as explained at the beginning of this chapter.

```
(defmac (andb  x y . z) @(bitop  1 ,x ,y . ,z))
(defmac (xorb  x y . z) @(bitop  6 ,x ,y . ,z))
(defmac (orb   x y . z) @(bitop  7 ,x ,y . ,z))
(defmac (norb  x y . z) @(bitop  8 ,x ,y . ,z))
(defmac (eqvb  x y . z) @(bitop  9 ,x ,y . ,z))
(defmac (notb  x)       @(bitop 12 ,x 0))
(defmac (nandb x y . z) @(bitop 14 ,x ,y . ,z))
(defmac (shlb  x y . z) @(bitop 16 ,x ,y . ,z))
(defmac (shrb  x y . z) @(bitop 17 ,x ,y . ,z))
(defmac (asrb  x y . z) @(bitop 18 ,x ,y . ,z))

(defun (andb  x y . z) (apply bitop  1 x y z))
(defun (xorb  x y . z) (apply bitop  6 x y z))
(defun (orb   x y . z) (apply bitop  7 x y z))
(defun (norb  x y . z) (apply bitop  8 x y z))
(defun (eqvb  x y . z) (apply bitop  9 x y z))
(defun (notb  x)       (bitop 12 x 0))
(defun (nandb x y . z) (apply bitop 14 x y z))
(defun (shlb  x y . z) (apply bitop 16 x y z))
(defun (asrb  x y . z) (apply bitop 17 x y z))
```

The **evenp** predicate returns truth, if its argument is divisible by two with a remainder of zero, i.e. if that argument is “even”. **Oddp** returns truth, if its argument is not even.

```
(defun (evenp x) (= 0 (rem x 2)))

(defun (oddp x) (not (evenp x)))
```

The **gcd** function computes the *greatest common divisor* of x and y using *Euclid's algorithm*. **Lcm** computes the least common multiple of x and y.

```
(defun (gcd x y)
  (defun (gcd x y)
    (cond ((= 0 x) y)
          ((= 0 y) x)
          ((< x y) (gcd x (rem y x)))
          (else    (gcd y (rem x y)))))
    (gcd (abs x) (abs y)))

(defun (lcm x y)
  (let ((cd (gcd x y)))
    (abs (* cd (div x cd) (div y cd)))))
```

The **expt** (“exponent”) function computes x to the power of y, where y has to be non-negative, because LISP9 has only fixnums. There are two different algorithms internally: a naive one (*nexpt*) that computes x^y by repeated multiplication and another one (*expt2*) that is optimized for large exponents and that is based on the observation that

$$x^n = (x^{\frac{n}{2}})^2 \quad \text{if } n \text{ is even}$$

Because the optimized algorithm reduces the number of multiplications from n to (ideally) $log(n)$, but also adds $2log(n)$ divisions, there is a rather high threshold for y where the optimization starts to pay off. For instance, at $y = 10$, there would be 10 multiplications in the naive algorithm and 5 multiplications

and 10 divisions in the optimized one. At $y = 20$, there are 6 steps in the optimized variant, giving 6 multiplications and 12 divisions.

The optimized variant makes much more sense in the presence of unbounded (bignum) integers. At $y = 100$, there would be 8 multiplications and 16 divisions, which is a great improvement over the naive approach.

```
(defun (expt x y)
  (defun (square x) (* x x))
  (defun (expt2 x y)
    (cond ((= 0 y) 1)
          ((evenp y)
            (square (expt2 x (div y 2))))
          (else
            (* x (square (expt2 x (div y 2)))))))
  (defun (nexpt y r)
    (cond ((= 0 y) r)
          (else (nexpt (- y 1) (* r x)))))
  (if (> y 20)
      (expt2 x y)
      (nexpt y 1)))
```

The **mod** function computes the *floored division remainder* $x - y \times \lfloor x/y \rfloor$ while the **rem** function computes the *truncated* remainder $x - y \times trunc(x/y)$. The notation $\lfloor x \rfloor$ denotes rounding toward $-\infty$ (i.e. the largest integer n such that $n \leq x$) and $trunc(x)$ denotes rounding toward zero (i.e. the cutting off of fractional digits).

x	y	mod	rem
17	5	2	2
17	−5	−3	2
−17	5	3	−2
−17	−5	−2	−2

Fig. 54 – Modulo and remainder

There is no difference between **rem** and **mod** as long as both of its operands have the same sign. See figure 54.

```
(defun (mod x y)
  (let ((r (rem x y)))
    (cond ((= 0 r) 0)
          ((eq (< x 0) (< y 0)) r)
          (else (+ y r)))))
```

21.9 String Functions

Most string functions are defined as primitive functions for the sake of efficiency. However, the **scopy** function, which creates a verbatim copy of its argument, is trivial to define in terms of **substr** and **ssize**.

```
(defun (scopy s)
  (substr s 0 (ssize s)))
```

21.10 Input/Output Functions

The **terpri** function sends a newline sequence to the specified port or to the current output port when no port is given. The optional port argument is stored in the list of optional arguments, p. When too many arguments are passed to **terpri**, it signals an error.

```
(defun (terpri . p)
  (if (and (pair p)
           (pair (cdr p)))
      (error "terpri: too many arguments"))
  (apply writec #\nl p))
```

The **print** function is a generalization of the **print()** function (page 125) that accepts any number of arguments.

```
(defun (print . xs)
  (cond ((null xs) (princ "\n"))
        ((null (cdr xs))
```

```
          (prin (car xs))
          (princ "\n"))
        (else
          (prin (car xs))
          (writec #\sp)
          (apply print (cdr xs)))))
```

The **readln** function reads one line of characters from the given port or from the current input port, if no port is specified. The line may be terminated by either a newline (**#\nl**) character or the end of the input stream. When the end of the input stream is reached *before* reading the first character, an EOF marker is returned.

```
(defun (readln . p)
  (let loop ((c (apply readc p))
             (a nil))
    (cond ((eofp c)
            (if (null a)
                c
                (liststr (nrever a))))
          ((c= #\nl c)
            (liststr (nrever a)))
          (else
            (loop (apply readc p)
                  (cons c a))))))
```

The **with-infile** and **with-outfile** functions redirect the *current input* and *output* port, respectively. They open the file s and connect it to the corresponding port. In the dynamic extent of that connection, they evaluate the *thunk* (nullary function) f. After evaluating f, they restore the port that was previously in effect (thereby terminating the dynamic extent of the connection) and return the result of (f).

with-infile and **with-outfile** are useful for redirecting the input and output of functions that operate on the current input and output port. For instance,

```
(with-infile "foo" readln)
```

would be almost equivalent to

```
(readln (open-infile "foo"))
```

Almost, because the **with-** functions close the redirected port before returning their result, while the above expression leaves the port connected to "foo" open.

```
(defun (with-infile s f)
  (let ((oi (inport))
        (i  (open-infile s)))
    (set-inport i)
    (let ((r (f)))
      (set-inport oi)
      (close-port i)
      r)))

(defun (with-outfile s f)
  (let ((oo (outport))
        (o  (open-outfile s)))
    (set-outport o)
    (let ((r (f)))
      (set-outport oo)
      (close-port o)
      r)))
```

The **with-inport** and **with-outport** functions are similar to **with-infile** and **with-outfile**, above, but instead of redirecting the current I/O ports, they pass a port object connected to the file s to the *unary* function f. The example in the previous paragraph could be written as follows using **with-inport**:

```
(with-inport "foo" readln)
```

The difference between them is that **readln** will be applied to a port argument when used with **with-inport**:

```
(with-infile "foo"          (with-inport "foo"
  (lambda ()                  (lambda (p)
    (readln)))                  (readln p)))
```

```
(defun (with-inport s f)
  (let* ((i (open-infile s))
         (r (f i)))
    (close-port i)
    r))

(defun (with-outport s f)
  (let* ((o (open-outfile s))
         (r (f o)))
    (close-port o)
    r))
```

21.11 Non-Local Exits

The **catch** function is like **catch***, but also registers its catch tag on the *unwind stack*, which is bound to the variable ***unwind***. The registration takes place before entering the dynamic extent of the catch form and is deleted after exiting from its dynamic extent, but before returning from **catch**.

The rationale behind the unwind mechanism is to be able to "undo" (or *unwind*) any changes applied to the state of a program inside of the dynamic extent of a catch form, so that the original state will be restored, no matter whether the catch context is exited by returning from its dynamic extent or by throwing the corresponding catch tag.

This is done by registering an unwind handler via **unwind**, which will be explained below.

```
(def *unwind* nil)

(defun (catch x)
  (let ((r (catch*
             (lambda (c)
```

```
                 (setq *unwind* (cons c *unwind*))
                 (x c)))))
    (setq *unwind* (cdr *unwind*))
    r))
```

The **throw** function is like **throw***, but also triggers evaluation of all *unwind handlers* that have been registered *after* establishing the catch tag c. **Throw** also removes unwind handlers from the unwind stack after evaluating them, but leaves the requested catch tag on top of the stack; it will be removed by the **catch** function.

Note that timing is critical when evaluating an unwind handler: The handler itself is removed from the stack *before* evaluating it. Otherwise performing a **throw** inside of the handler would result in infinite recursion.

```
(defun (throw c v)
  (let loop ()
    (cond ((null *unwind*)
            (throw* c v))
          ((funp (car *unwind*))
            (let ((w (car *unwind*)))
              (setq *unwind* (cdr *unwind*))
              (w)
              (loop)))
          ((eq c (car *unwind*))
            (throw* c v))
          (else
            (setq *unwind* (cdr *unwind*))
            (loop)))))
```

The **unwind** form **(unwind u f)** is used to make sure that the effect of the *unwind handler* u takes place whenever the flow of control passes from the dynamic extent of f to some point outside of the dynamic extent of f. This is true, in particular, when a **throw** form inside of f is caught by a **catch** form outside of the dynamic extent of f.

In the trivial case, where no non-local transfer of control takes place, **unwind** first evaluates (f), then evaluates (u) and finally returns the value of (f):

```
(unwind (lambda () (print "unwind"))
        (lambda () (print "fun")
                   'result))
```

will first print **fun**, then print **unwind**, and finally return **result.**

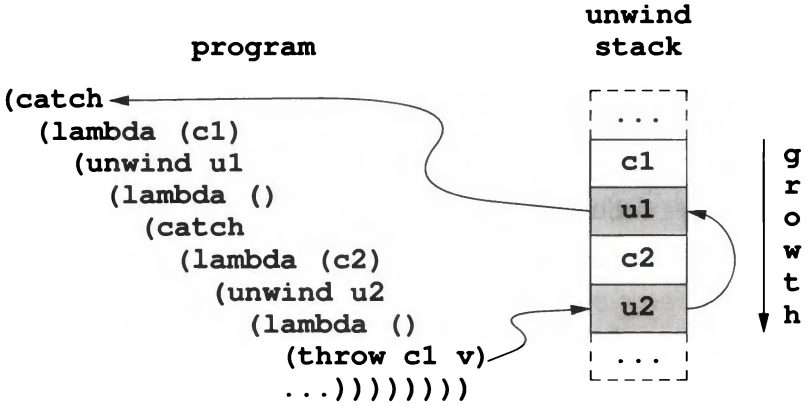


Fig. 55 – Unwinding the unwind stack

Multiple unwind handlers may be registered in the dynamic extents of different catch tags or even in the dynamic extent of the same catch tag. In either case, *all* unwind handlers that have been registered (and not yet unregistered) between a throw form and the corresponding catch form will evaluate when control passes "upward" (toward **catch**) and across an **unwind** form. See figure 55, where first $u2$ and then $u1$ will be triggered by **throw**.

When an unwind form is exited "normally" (i.e. not through **throw**), it will unregister u before returning. Therefore, when an unwind form is *completed* inside of the dynamic extent of a catch form, then throwing the corresponding catch tag will *not* trigger the unwind handler of the unwind form. In the program

```
(catch
  (lambda (ct)
    (unwind u1 f1)
    (unwind u2
```

```
      (lambda ()
        (throw ct v)))))
```

throwing *ct* will trigger evaluation of the unwind handler *u*2, but not the evaluation of *u*1, because evaluation of the first unwind form is already completed and *u*1 already has been evaluated.

Finally note that the **with** special form uses an unwind form internally to undo all modifications made by it. Hence the following program will not have any surprising effect:

```
(def *who* 'unknown)

(defun (hello) (print 'hello *who*))

(catch
  (lambda (ct)
    (with ((*who* 'somebody))
      (throw ct 'decided-otherwise))))
```

That is, it will reset the value of ***who*** to **unknown**, even though the **with** form is exited through a non-local exit.

```
(defun (unwind u f)
  (setq *unwind* (cons u *unwind*))
  (let ((v (f)) (w nil))
    (setq w (car *unwind*))
    (setq *unwind* (cdr *unwind*))
    (w)
    v))
```

The **catch-errors** special form creates a catch tag that will be thrown when an error is signalled by the LISP9 system while evaluating the body of the form. The form uses **catch*** (not **catch**!) internally and expands as follows:

```
(catch-errors (v) x ···)
        →  (let ((G3 *Errtag*)
                 (G4 *Errval*))
             (let ((G2 (catch
                          (lambda (G1)
                            (setq *Errval* v)
```

```
                          (setq *Errtag* G1)
                          x ···))))
              (setq *Errtag* G3)
              (setq *Errval* G4)
              G2))
```

When no value v is being passed to the special form, as in

```
(catch-errors () x ···)
```

then ***Errval*** will also be set to G_1, indicating that the error message passed to the error handler shall be used as the error value. For instance:

```
(catch-errors () (car nil))
                    =>  "car: expected pair"
```

The **catch-errors** form serves to handle or ignore error conditions. For instance:

```
(delete foo)
```

will signal an error, if the file specified in *foo* does not exist. Sometimes it may be useful to just make sure that a files does not exist. This can be done in two ways:

```
(if (exists foo) (delete foo))
```

works, but might fall prey to a race condition where the file is deleted by a different process after the evaluation of **exists** but before the evaluation of **delete**. So the proper way to do this would be

```
(catch-errors () (delete foo))
```

Under some circumstances, it might be useful to deliver a specific value in a case where an error would normally be signalled. For instance, in some computations returning zero for $x/0$ may make sense:

```
(catch-errors (0) (div x y))
```

The following form would make **car** return **nil** when applied to an atom:

```
(lambda (x) (catch-errors (nil) (car x)))
```

However, this is probably better implemented using an **if** form, because **catch-errors** is a rather expensive operation.

```
(defmac (catch-errors v x . xs)
  (let ((g  (gensym))
        (r  (gensym))
        (et (gensym))
        (ev (gensym)))
    @(let ((,et *Errtag*)
           (,ev *Errval*))
       (let ((,r (catch*
                   (lambda (,g)
                     (setq *Errval*
                           ,(if (null v)
                                g
                                (car v)))
                     (setq *Errtag* ,g)
                    ,x . ,xs))))
         (setq *Errtag* ,et)
         (setq *Errval* ,ev)
         ,r))))
```

21.12 Image Files

The **save** function dumps the current state of the LISP9 system to the image file from which the state was loaded initially, i.e. it updates the current image file. When no image was loaded (which normally happens only when bootstrapping LISP9), the user is advised to use **dump-image** instead.

```
(defun (save)
  (if *imagefile*
      (dump-image *imagefile*)
      (error
        "save: no image loaded, use \"dump-image\"")))
```

21.13 Built-In Functions

The following definitions create **apply**-able functions that correspond to inlined primitive functions. See page 346 for an explanation. The first three thunks of definitions will create nullary, unary, and binary functions. These are straightforward definitions without anything special going on.

```
(defun (cmdline) (cmdline))
(defun (errport) (errport))
(defun (inport) (inport))
(defun (outport) (outport))
(defun (gc) (gc))
(defun (gensym) (gensym))
(defun (obtab) (obtab))
(defun (quit) (quit))
(defun (symtab) (symtab))

(defun (abs x) (abs x))
(defun (alphac x) (alphac x))
(defun (atom x) (atom x))
(defun (catch* x) (catch* x))
(defun (char x) (char x))
(defun (charp x) (charp x))
(defun (charval x) (charval x))
(defun (close-port x) (close-port x))
(defun (ctagp x) (ctagp x))
(defun (constp x) (constp x))
(defun (delete x) (delete x))
(defun (downcase x) (downcase x))
(defun (dump-image x) (dump-image x))
(defun (eofp x) (eofp x))
(defun (existsp x) (existsp x))
(defun (fixp x) (fixp x))
(defun (format x) (format x))
(defun (funp x) (funp x))
(defun (inportp x) (inportp x))
(defun (liststr x) (liststr x))
```

```
(defun (listvec x) (listvec x))
(defun (load x) (load x))
(defun (lowerc x) (lowerc x))
(defun (mx x) (mx x))
(defun (mx1 x) (mx1 x))
(defun (not x) (not x))
(defun (null x) (null x))
(defun (numeric x) (numeric x))
(defun (open-infile x) (open-infile x))
(defun (outportp x) (outportp x))
(defun (pair x) (pair x))
(defun (set-inport x) (set-inport x))
(defun (set-outport x) (set-outport x))
(defun (ssize x) (ssize x))
(defun (stringp x) (stringp x))
(defun (strlist x) (strlist x))
(defun (symbol x) (symbol x))
(defun (symbolp x) (symbolp x))
(defun (symname x) (symname x))
(defun (syscmd x) (syscmd x))
(defun (untag x) (untag x))
(defun (upcase x) (upcase x))
(defun (upperc x) (upperc x))
(defun (veclist x) (veclist x))
(defun (vectorp x) (vectorp x))
(defun (vsize x) (vsize x))
(defun (whitec x) (whitec x))

(defun (div x y) (div x y))
(defun (eq x y) (eq x y))
(defun (nreconc x y) (nreconc x y))
(defun (rem x y) (rem x y))
(defun (reconc x y) (reconc x y))
(defun (setcar x y) (setcar x y))
(defun (setcdr x y) (setcdr x y))
(defun (sfill x y) (sfill x y))
(defun (sref x y) (sref x y))
(defun (throw* x y) (throw* x y))
```

```
(defun (vfill x y) (vfill x y))
(defun (vref x y) (vref x y))
```

The LSUBRs fold their functions over their argument lists. Their functions have to be *eta-expanded* in order to avoid inlining. E.g.

```
(fold + 0 x)
```

would not work, but after eta-expanding **+**, it will. Eta-expansion is a transformation that turns a term into an abstraction, i.e.

```
M → (lambda (x) (M x))
```

The transformation does not affect the meaning of the term in normal-order lambda calculus, but is sometimes used in applicative-order lambda calculus to avoid premature reduction of a form [Holm2016]. In the LISP9 library, it is used to defer inlining of primitive functions, so they can be passed to higher-order functions.

```
(defun (+ . x)
  (fold (lambda (x y) (+ x y)) 0 x))

(defun (* . x)
  (fold (lambda (x y) (* x y)) 1 x))

(defun (conc . x)
  (fold (lambda (x y) (conc x y)) nil x))

(defun (nconc . x)
  (fold (lambda (x y) (nconc x y)) nil x))

(defun (sconc . x)
  (fold (lambda (x y) (sconc x y)) "" x))

(defun (vconc . x)
  (fold (lambda (x y) (vconc x y)) #() x))
```

The OSUBRs with one optional argument apply their (inline) functions to the argument, if one is given. They signal an error

when too many arguments are passed to them.

```
(defun (peekc . x)
  (cond ((null x)
          (peekc))
         ((null (cdr x))
          (peekc (car x)))
         (else
          (error "peekc: too many arguments"))))

(defun (read . x)
  (cond ((null x)
          (read))
         ((null (cdr x))
          (read (car x)))
         (else
          (error "read: too many arguments"))))

(defun (readc . x)
  (cond ((null x)
          (readc))
         ((null (cdr x))
          (readc (car x)))
         (else
          (error "readc: too many arguments"))))
```

The **bitop** function uses an additional mandatory argument (and conses it to its operand list) in order to catch the case where it is applied to an insufficient number of arguments.

```
(defun (bitop op x y . z)
  (fold (lambda (x y) (bitop op x y))
        x
        (cons y z)))
```

The **-** function distinguishes two cases: negation of a single argument and subtraction of any larger number of arguments.

```
(defun (- x . y)
  (if (null y)
      (- x)
      (fold (lambda (x y) (- x y)) x y)))
```

The **min** and **max** functions use their first argument (x) as the neutral (default) element when folding over the remaining arguments (y).

```
(defun (max x . y)
  (if (null y)
      x
      (fold (lambda (x y) (max x y)) x y)))

(defun (min x . y)
  (if (null y)
      x
      (fold (lambda (x y) (min x y)) x y)))
```

The OSUBRs with one mandatory argument use the same method as **bitop**, above, to make sure that a sufficient number of arguments is supplied.

```
(defun (error x . y)
  (cond ((null y)
          (error x))
        ((null (cdr y))
          (error x (car y)))
        (else
          (error "error: too many arguments"))))

(defun (mkstr x . y)
  (cond ((null y)
          (mkstr x))
        ((null (cdr y))
          (mkstr x (car y)))
```

```
        (else
          (error "mkstr: too many arguments"))))

(defun (mkvec x . y)
  (cond ((null y)
          (mkvec x))
        ((null (cdr y))
          (mkvec x (car y)))
        (else
          (error "mkstr: too many arguments"))))

(defun (numstr x . y)
  (cond ((null y)
          (numstr x))
        ((null (cdr y))
          (numstr x (car y)))
        (else
          (error "numstr: too many arguments"))))

(defun (strnum x . y)
  (cond ((null y)
          (strnum x))
        ((null (cdr y))
          (strnum x (car y)))
        (else
          (error "strnum: too many arguments"))))

(defun (open-outfile x . y)
  (cond ((null y)
          (open-outfile x))
        ((null (cdr y))
          (open-outfile x (car y)))
        (else
          (error
            "open-outfile: too many arguments"))))

(defun (prin x . y)
  (cond ((null y)
```

```
          (prin x))
        ((null (cdr y))
          (prin x (car y)))
        (else
          (error "prin: too many arguments"))))

(defun (princ x . y)
  (cond ((null y)
          (princ x))
        ((null (cdr y))
          (princ x (car y)))
        (else
          (error "princ: too many arguments"))))

(defun (writec x . y)
  (cond ((null y)
          (writec x))
        ((null (cdr y))
          (writec x (car y)))
        (else
          (error "writec: too many arguments"))))
```

The **%compare** function is an internal predicate that iterates over the list a and makes sure that $(op\ a_i\ a_{i+1})$ holds for any two subsequent elements a_i, a_{i+1} of a. The condition is trivially fulfilled, if a has only one single element. The predicate returns truth, if the condition is satisfied.

%Compare is used internally to implement all OSUBR comparison functions, like **=**, **<**, **c=**, **ci>=**, **s<**, **s>**, etc. For instance, the **=** predicate would be implemented as

```
(defun (= x . y)
  (%compare (lambda (x y) (= x y)) (cons x y)))
```

This approach poses one problem, though: whenever the function **%compare** is redefined in a program, all of the below predicates will no longer work. This problem is mitigated weakly here by including the special character "**%**" in the name of the function in the hope that no user-level code will use it.

A safer (but also more memory-intensive) solution would be to make **%compare** a higher-order function that returns the desired comparison function:

```
(defun (compare op)
  (lambda (x . y)
    (let loop ((a (cons x y)))
      (cond ((null (cdr a)))
            ((op (car a) (cadr a))
              (loop (cdr a)))
            (else nil))))))
```

The = predicate, for example, could then be defined as

```
(def = (compare (lambda (x y) (= x y))))
```

However, there are many OSUBR comparison predicates in LISP9 and hence the former approach is chosen here. When memory is not a critical constraint, the below functions can be replaced with definitions using **compare** without any side effects.

```
(defun (%compare op a)
  (let loop ((a a))
    (cond ((null (cdr a)))
          ((op (car a) (cadr a))
            (loop (cdr a)))
          (else nil))))

(defun (< x . y)
  (%compare (lambda (x y) (< x y)) (cons x y)))

(defun (<= x . y)
  (%compare (lambda (x y) (<= x y)) (cons x y)))

(defun (= x . y)
  (%compare (lambda (x y) (= x y)) (cons x y)))

(defun (> x . y)
  (%compare (lambda (x y) (> x y)) (cons x y)))
```

```
(defun (>= x . y)
  (%compare (lambda (x y) (>= x y)) (cons x y)))

(defun (c< x . y)
  (%compare (lambda (x y) (c< x y)) (cons x y)))

(defun (c<= x . y)
  (%compare (lambda (x y) (c<= x y)) (cons x y)))

(defun (c= x . y)
  (%compare (lambda (x y) (c= x y)) (cons x y)))

(defun (c> x . y)
  (%compare (lambda (x y) (c> x y)) (cons x y)))

(defun (c>= x . y)
  (%compare (lambda (x y) (c>= x y)) (cons x y)))

(defun (s< x . y)
  (%compare (lambda (x y) (s< x y)) (cons x y)))

(defun (s<= x . y)
  (%compare (lambda (x y) (s<= x y)) (cons x y)))

(defun (s= x . y)
  (%compare (lambda (x y) (s= x y)) (cons x y)))

(defun (s> x . y)
  (%compare (lambda (x y) (s> x y)) (cons x y)))

(defun (s>= x . y)
  (%compare (lambda (x y) (s>= x y)) (cons x y)))

(defun (si< x . y)
  (%compare (lambda (x y) (si< x y)) (cons x y)))

(defun (si<= x . y)
  (%compare (lambda (x y) (si<= x y))
```

```
               (cons x y)))

(defun (si= x . y)
  (%compare (lambda (x y) (si= x y)) (cons x y)))

(defun (si> x . y)
  (%compare (lambda (x y) (si> x y)) (cons x y)))

(defun (si>= x . y)
  (%compare (lambda (x y) (si>= x y))
               (cons x y)))
```

There are no surprises in the definitions of the ternary built-in functions.

```
(defun (sset x y z) (sset x y z))
(defun (substr x y z) (substr x y z))
(defun (subvec x y z) (subvec x y z))
(defun (vset x y z) (vset x y z))
```

The **apply** function conses all its optional arguments (x_i) to the last argument (a) before passing the argument list to the **apply** special form. I.e.

(apply f x_1 $\cdots$ x_n a**)**

becomes

(apply f **(cons** x_1

$\cdots$

(cons x_n a**)** $\cdots$ **))**

This is the same transformation as performed by the compiler when compiling the primitive (inlined) **apply** form.

```
(defun (apply f . as)
  (let loop ((as as)
             (a  nil))
    (cond ((null as)
```

```
          (error "apply: too few arguments"))
        ((null (cdr as))
          (if (pair (car as))
              (apply f (conc (nrever a)
                             (car as)))
              (error "apply: expected list"
                     (car as))))
        (else
          (loop (cdr as)
                (cons (car as) a))))))
```

PART III
ADDENDA

22. A Sample Run

The following code code snippets will show the definition of the **fold** function (page 365) of the LISP9 library in the various stages of transformation before it is being interpreted. Some of the stages can be made visible from within the LISP9 system, some of them have to extracted by hacking the C part of the system.

Here is the definition of **fold**, as contained in the library:

```
(defun (fold f b a)
  (defun (fl a r)
    (if (null a)
        r
        (fl (cdr a)
            (f r (car a)))))
  (fl a b))
```

The first step in the transformation is the expansion of the embedded **defun** form to a **labels** special form. This transformation is an integral part of the macro expansion step, and the result of this step cannot be made visible at the LISP9 prompt. To retrieve this intermediate form, the result of the **expandbody()** function in the LISP9 code has be printed.

```
(defun (fold f b a)
  (labels ((fl (lambda (a r)
                 (if (null a)
                     r
                     (fl (cdr a)
                         (f r (car a)))))))
    (fl a b)))
```

In the next step, the outer **defun** form is transformed to a combination of **def** and **lambda**. This transformation is also performed internally during macro expansion. It is the result of the **expanddef()** function.

```
(def fold
  (lambda (f b a)
    (labels ((fl (lambda (a r)
```

```
                    (if (null a)
                        r
                        (fl (cdr a)
                            (f r (car a)))))))
     (fl a b))))
```

The next two macro expansion steps can be replicated at the LISP9 prompt. The first step is performed by applying the **mx1** function to the above intermediate form. This step expands the **labels** special form to **let** and **setq**:

```
(def fold
  (lambda (f b a)
    (let ((fl nil))
      (setq fl (lambda (a r)
                 (if (null a)
                     r
                     (fl (cdr a)
                         (f r (car a))))))
      (fl a b))))
```

The next step transforms the **let** special form to an application of a lambda function. Immediately after retrieving the above intermediate form, the next form can be generated by evaluating the expression **(mx1 **)**. The result of this step is the fully macro-expanded form of the original expression and hence the form that will be compiled subsequently.

```
(def fold
  (lambda (f b a)
    ((lambda (fl)
       (setq fl (lambda (a r)
                  (if (null a)
                      r
                      (fl (cdr a)
                          (f r (car a))))))
       (fl a b))
     nil)))
```

The first compilation step is closure conversion. To retrieve the below form, the result of the **clsconv()** function has to be printed

inside of **eval()** in the compiler source code. This step performs the following transformations:

- it replaces **def** with **setq**
- it replaces free variable names with references to environment slots (**%ref** forms)
- it replaces bound variable names with references to argument slots (**%arg** forms)
- it replaces **lambda** special forms with **%closure** forms and inserts init maps after the argument lists of closures

The below code will store the function created by the outermost **%closure** form in slot 46 of the top-level environment. The slot is assigned to the variable *fold* during closure conversion.

```
(setq (%ref 46 fold)
  (%closure (f b a) nil
    ((%closure (fl) ((a 0 0 f)
                     (a 2 1 a)
                     (a 1 2 b))
       (setq (%arg 0)
         (%closure (a r) ((a 0 0 fl)
                          (e 0 1 f))
           (if (null (%arg 0))
               (%arg 1)
               ((%ref 0 fl)
                 (cdr (%arg 0))
                 ((%ref 1 f) (%arg 1)
                             (car (%arg 0))))))))
       ((%arg 0) (%ref 1 a) (%ref 2 b)))
     nil)))
```

The final step of the compilation process is bytecode generation. The bytecode of a function can be extracted using the **untag** function. **(Untag fold)** would return the below string containing the bytecode of the function:

```
"\15\0\206\30\0\3\5\0\0\10\n\0\1\15\0o\30\0\1\15
\0H\30\0\2\6\0\0[\16\0&\6\0\1\15\0G\6\0\0$\10\6
\0\1\10\n\0\2\7\0\1\0\260\3\10\6\0\0'\10\n\0\2\7
```

```
\0\0\0\313\4\32\24\0\2\27\0\0\0\0\26\0\0\0\1\23
\0\26\33\0\0\7\0\2\0\261\10\7\0\1\0\1\10\n\0\2\6
\0\0\4\32\24\0\3\27\0\0\0\0\27\0\2\0\1\27\0\1\0
\2\23\0\20\4\32\25\23\0\3\34\0.\20"
```

Of course, the bytecode is not exactly human-readable, but the disassembler can help to make it more comprehensible. In case the disassembler has been loaded (**,ldisasm**) or is contained in the LISP9 image file, the above code can be disassembled by evaluating the expression

```
(disasm fold)
```

The output of the disassembler is shown on the following page in two-column format, i.e. first read the left column and then the right column.

```
  0 jmp     134
  3 enter   3
  6 quote   nil
  9 push
 10 pushval 1
 13 jmp     111
 16 enter   1
 19 jmp     72
 22 enter   2
 25 arg     0
 28 null
 29 brf     38
 32 arg     1
 35 jmp     71
 38 arg     0
 41 car
 42 push
 43 arg     1
 46 push
 47 pushval 2
 50 ref     1 f
 55 apply
 56 push
 57 arg     0
 60 cdr
 61 push
 62 pushval 2
 65 ref     0 fl
 70 tailapp
 71 return
 72 mkenv   2
 75 cparg   0 0
 80 cpref   0 1
 85 closure 22
 88 setarg  0
 91 ref     2 b
 96 push
 97 ref     1 a
102 push
103 pushval 2
106 arg     0
109 tailapp
110 return
111 mkenv   3
114 cparg   0 0
119 cparg   2 1
124 cparg   1 2
129 closure 16
132 tailapp
133 return
134 propenv
135 closure 3
138 setref  46
141 halt
```

23. Performance

The LISP9 interpreter is pretty fast for a stack machine-based bytecode interpreter. The table in figure 56 compares the run time of the widely used *BOYER benchmark* test in various Scheme systems as well as LISP9. The BOYER benchmark was considered to be one of the more realistic benchmark tests in [Gabriel1986].

System	Version	Type	Time	Perf.
(Compiler)	–	native	< 1s	0.09
LISP9	20190311	bytecode	11.6s	1.00
Chicken Scheme	4.13.0	bytecode	12.3s	1.06
S9fES	20181205	bytecode	13.6s	1.17
MIT Scheme	9.2	bytecode	24.5s	2.11
S9fES (old)	20180823	tree walker	57.7s	4.97

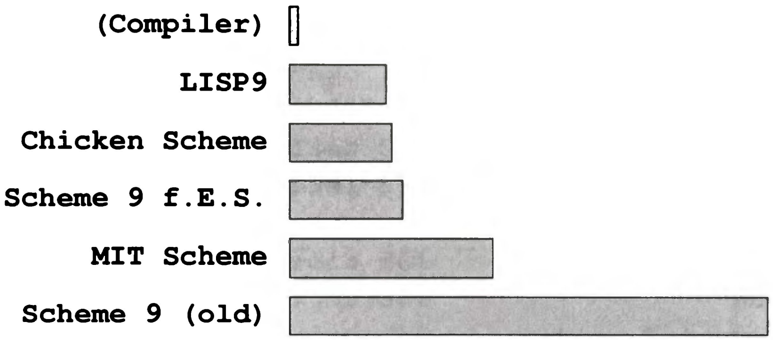


Fig. 56 – BOYER benchmark results

In the figure, "native" denotes a native code compiler, "bytecode" indicates a bytecode compiler and interpreter, "tree walker" indicates an interpreter that processes the S-expression representation of a program (as returned by **read**) directly. The run times are given in seconds and "perf." ("performance") indicates the run time relative to the fastest bytecode-based system in the test. Smaller values indicate faster evaluation.

For reference, the comparison also includes compilation to native code, which both MIT Scheme [MIT-Scheme] and Chicken

Scheme [Chicken] provide. Both compilers generate code that evaluates the Boyer test in less than a second. Tests were performed on the author's 750MHz notebook computer.

Another interesting detail is the memory footprint of the compiled BOYER program, which consists of about 550 lines of Scheme or LISP9 code. When loading the program into the Scheme 9 system [S9fES], it will be compiled to a program with inlined object references, as discussed on pages 186+, while in LISP9 it will be compiled to bytecode with a separate object table.

The program allocates 5911 nodes and 26818 vector cells in Scheme 9 (giving about 259K bytes) and 2697 nodes and 1703 vector cells in LISP9 (giving about 26K bytes). So maintaining an object table indeed decreases the memory footprint of the bytecode by an order of magnitude in this case.

24. Aspects of Compilation

24.1 Compiling LISP-N

LISP9 is a *LISP-1*, a LISP system with a single namespace for all objects that appear in the system (except for macros, which live in a compile-time environment). Other LISP languages, like Common Lisp, provide multiple name spaces for different types of objects, like data objects, functions, and label names in tag bodies [CLtL1984]. Such systems are often referred to as *LISP-2* (two-namespace) or *LISP-N* (multiple-namespace) languages.

In principle, compiling bytecode for such systems is not much different from compiling code for a LISP-1. The basic idea of a LISP-2 is that function objects are contained in a different "name space" (which *may* be thought of as an environment) than other data objects. The advantage of this approach is that the same name can be used for functions and data without confusing them. E.g., Common Lisp allows you to write:

```
(defun flatten (list)
  (cond ((null list) nil)
        ((atom list) (list list))
        (t (append (flatten (car list))
                   (flatten (cdr list))))))
```

rather than making you invent a different name for the *list* argument. Each **list** in **(list list)** is unambiguous, because the first element in a function call will be looked up in the "function" namespace and its other elements in the "variable" namespace.

The disadvantage of the two-namespace approach is that variables will not be recognized in operator positions and functions will not be recognized in operand positions without explicit references. Compare, for instance the LISP-2 code

```
(defun compl (p)
  (lambda (x) (not (funcall p x))))

(funcall (compl (function atom)) 'foo)
```

to the LISP-1 program

```
(defun (compl p)
  (lambda (x) (not (p x))))

((compl atom) 'foo)
```

The **function** and **funcall** operators are easy to implement once a mechanism for separate name spaces is in place. When no operator is given, the compiler will look up the *function* in

```
(function argument ...)
```

in the function namespace and each *argument* in the variable namespace. The **funcall** operator makes it look up *function* in the variable namespace and **function** looks up an *argument* in the function name space.

An obvious idea to implement multiple namespaces would be to actually carry around multiple environments. However, doing to will increase the complexity of both the compiler and the abstract machine. It might even degrade the performance of the system, because multiple environment pointers have to be maintained during function application. While this approach might be manageable in a LISP-2 environment, it certainly becomes impractical in a LISP-N for $n > 2$.

A different approach would be to attach a hidden and unreadable prefix to each symbol that indicates its namespace. The prefix has to be unreadable by **read**, or prefixes could be generated accidentally – or even deliberately – in programs. When using prefixes, the above example would translate internally to something like

```
(defun fun:compl (var:p)
  (lambda (var:x) (not (funcall var:p var:x))))
```

(assuming that **:** is unreadable). In this case, the **function** and **funcall** operators merely switch the prefix of the symbol to look up.

Note that the **funcall** in **(funcall var:p var:x)** does not change the prefix of *var:p*, because it already has the proper prefix. If *p*

appeared in an operator position, the compiler would translate it to *fun:p*, which the **funcall** operator would then change to *var:p*.

Also note that the **funcall** in

```
(funcall (compl (function atom)) 'foo)
```

does not involve any variable lookup, but is merely an idiosyncrasy resulting from the internals of some LISP systems. Exploring such details is left to the interested reader.

24.2 Stand-Alone Compilation

When compiling to a stand-alone executable, be it native code or bytecode packaged with an interpreter, things are a bit different. In fact they are *simpler*. Many parts of the LISP9 system have been designed specifically to deal with a dynamically changing environment. These parts can be simplified or even omitted when generating an executable file.

The major difference between interactive compilation and *non-interactive compilation* (*batch compilation*) is that usually no redefinitions of top-level variables take place. Once a function has been compiled or a data object has been created, its lifetime can be assumed to be indefinite. I.e., the following happens pretty much *never* in batch-compiled code:

(defun (*f* ···**)** $body_1$**)**

expression ···

(defun (*f* ···**)** $body_2$**)**

In this scenario the body of the first function *f* would become garbage as soon as a new definition for *f* is being compiled. This is the point where a lot of complexity is added to an interactive system, because properly disposing of $body_1$ involves recycling all data objects referred to in it as well as recycling the vector holding its bytecode. Without these mechanisms, the node and vector pool of the LISP9 system would slowly fill up with dead functions and data objects during interactive program development, where the redefinition of functions happens all the time.

The mechanisms that deal with the disposal of dead functions are

- object table management (page 43, 186)
- the closure object format (page 286)

Object table management can be eliminated completely when compiling to a stand-alone executable. That is, the object table still has to be built, but its slots can be expected to contain the same object during the lifetime of the program, so the entire procedure of identifying unused slots and garbage-collecting them is not necessary.

The closure object, which is the internal representation of a compiled function, holds a byte vector containing the code of the function. The only reason for this vector to exist is the proper disposal of dead code when the closure becomes garbage. When batch-compiling, there is no need to dispose of dead functions, because functions will never become garbage.

Of course, this is not entirely true. There may be some corner cases in some programs where functions become deadweight at some point. For instance, a program may contain two versions of the same function for the sake of debugging. In this case, however, rather than garbage-collecting the unused function, it would make more sense to detect that it is not being used at all and hence not compile it in the first place. I.e., in

```
(defun (f ···) body1)
; f is unused here
(defun (f ···) body2)
```

the compiler should recognize that the first instance of f is never used and skip its compilation entirely. Another corner case would be the one where f is being used between its first and second definition, i.e.:

```
(defun (f ···) body1)
(f ···)
(defun (f ···) body2)
```

In this case the compiler would indeed carry around a dead function after its redefinition. This would be acceptable, though, because it is a highly improbable scenario and one might even

argue that it is a bug, because the name of the function f is reused after its first instance has performed some one-time action, like the initialization of a data structure. If the function name was not reused, the code of the function f would have to be carried along anyway. After all, all initialization functions become dead weight once their task is done. Therefore, a programmer should not expect the body of the first f to be recycled in this case.

When functions no longer need to be garbage collected, all bytecode can be stored in a single vector. See figures 57 and 58.

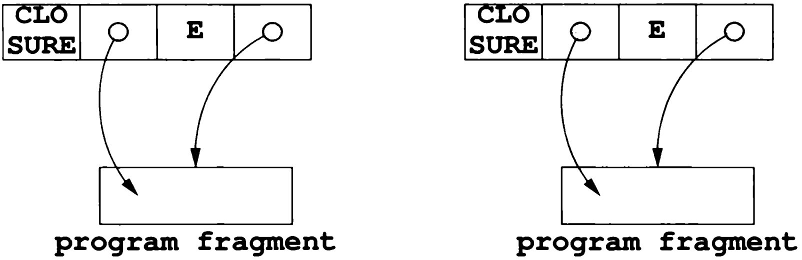


Fig. 57 – Bytecode in a dynamic environment

In a dynamic environment, each closure points to its own program fragment and its entry point is an offset into that fragment. In a stand-alone environment, there is one single bytecode vector and the entry point of each closure is just an offset into that vector.

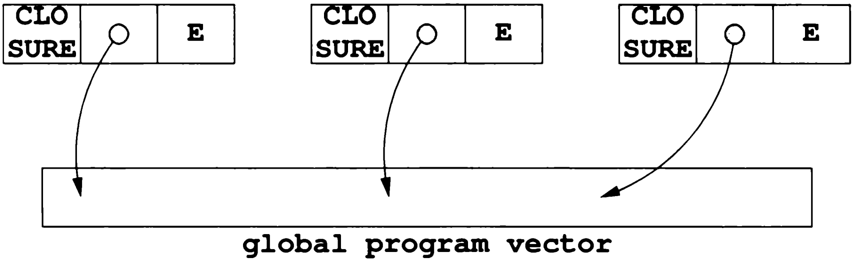


Fig. 58 – Bytecode in a stand-alone environment

Beside the simplified layout the closure object, there is one additional advantage to this approach: the *apply* instruction of the abstract machine no longer has to save the tuple $(Ip+1, Prog)$ when transferring control to a function. The value $Ip+1$ will do, because the program space is now the same for all functions.

In this case the simplification comes at a cost, though. Because all

programs share the same byte vector, they also share the same address space. Therefore, the 16-bit inline operands of the LAM may no longer be big enough. Increasing the operand size from two to three bytes increases the address space from $65,536$ to $16,777,216$ bytes, which should be sufficient to compile even the most complex programs.

While the LISP9 compiler and the LAM have been constructed in such a way that this transition should be manageable, there are quite a few points in the code that have to be adjusted manually. Most of them can be found in the garbage collection part of object table management and, of course, in the LISP9 abstract machine itself. Most prominently, the instruction format would have to change accordingly. For instance, the *jmp n m* instruction would become *jmp k n m* and its inline operand would be interpreted as $65536k + 256n + m$.

24.3 Compiling to Machine Code

24.3.1 Architectural Differences

The *LISP9 Abstract Machine* (*LAM*) is a stack-based architecture, while most hardware CPUs are register-based architectures. One simple approach for interpreting *stack machine* code on a register-based CPU is to open-code (inline) program fragments that resemble stack machine instructions. For instance:

```
LAM         Machine Code     Comment

quote 2     MOV  A,2         Load accumulator with 2
push        PUSH A           Push accumulator
arg   1     MOV  A,FP(1)     Load acc. with 1st arg
times       POP  B           Pop auxilary register
            MUL  A,B         A times B, result in A
```

However, the drawbacks of this approach are easily visible. For every two-argument instruction, the first operand has to be transferred to the stack and then back to an auxiliary register, while it would be much more efficient to store it in a register immediately. Fortunately, the conversion from stack machine code to register machine code is a well-understood process. Describing this process is beyond the scope of this text, though. The

interested reader may refer to [Holm1996]. For the optimization of stack machine code, see [Koopman1994]. Register allocation schemes for stack machines are described in [Shannon2006].

24.3.2 Object Table Management

Depending on the target architecture, scanning the machine code for references to literal data objects (see page 44) may be more or less complicated. When the target is a highly regular RISC architecture, a register (*OBTAB*) can be allocated globally to point to the object table. When all instructions have the same length and register names can easily be identified in instructions, the object marking routine may indeed scan the native machine code for references. For instance, the code loading a reference to the object in object table slot n may look like the following (ignoring the necessary scaling of n):

```
MOV  B,OBTAB
ADD  B,n
MOV  A,(B)
```

When register names can be identified easily in the MOV instruction, the **marklit()** function (page 45) can be rewritten to mark literal object references in native machine code. It would identify the “move” instruction moving *OBTAB* to B, go to the next instruction and extract the operand n, which would be the offset of the object table slot to mark.

When the target has a highly irregular instruction set, though, like the popular x86 CPUs, then this approach is completely impractical. In this case, an instruction loading a reference to the AST from which the machine code was compiled could be stored at the beginning of the machine code, i.e.:

```
MOV  A,address of AST
```

In this case, the garbage collector would only have to "know" where the address is located in that initial load instruction. When executing the code, the additional MOV instruction will not cause any harm. To save space, the AST could be harvested for data objects and the rest of the AST could be discarded. E.g.,

```
(cons (+ a 1) nil)
```

would reduce to

```
(1 nil)
```

24.3.3 Downward-Growing Stacks

The stack of the LAM grows “upward” (the next element to be pushed to the stack gets a higher address than the current top of the stack) while most hardware CPUs use “downward” growing stacks (the next element gets a lower address).

This is easily compensated for by reversing the layout of the stack frame (figure 34, page 173) and changing the sign in instructions that reference the stack. For instance, the *arg* instruction (page 175) would have to change from

$$Acc \leftarrow \mathbf{S}[[Fp - (256n + m)]]$$

to

$$Acc \leftarrow \mathbf{S}[[Fp + (256n + m)]]$$

before translating it to machine code.

24.3.4 Machine Registers

Machine code implementing the LAM requires the following registers

- an accumulator holding (intermediate) values
- at least one auxiliary register for binary operations
- an instruction pointer
- a stack pointer
- a frame pointer
- an object table pointer
- an environment pointer
- in a dynamic environment, a program fragment pointer

When the hardware provides those, the native stack and frame pointer can be used. In this case the garbage collector has to mark objects on the hardware stack, though, so it has to "know"

the anatomy of a function call on the target CPU in order to exclude return addresses and saved stack frame addresses (which are both unboxed) from the collection.

When the target has fewer registers than required by the LAM, the least frequently used registers can be spilled to memory. This would be the object table pointer, environment pointer, and program fragment pointer.

When the target has plenty of registers, it makes sense to use them. This can be done by caching the top of the stack [Holm1996] or by adopting more elaborate global register allocation strategies [Shannon2006].

24.3.5 Generating Core Instructions

Some of the core instructions of the LAM are so complex that they are best packaged in a routine and called from the generated code. Implementations of such instructions can be derived from their formal descriptions (pages 173+) or directly copied from the code of the LISP9 system (pages 282+). The instructions that are too complex for open-coding them are these:

applis, *applist*, *tailapp* (complex case), *catch**, *throw**, *entcol*, *mkenv*, *closure*, *macro*.

For the rest of the LAM instructions, machine code instructions can be *open-coded*, i.e. directly inserted into the target program. Most LAM instructions translate to native machine code pretty directly. For instance, the *apply* instruction (page 177), which is defined as follows

$$Ep\downarrow;\ \ (Ip+1,\ Prog)\downarrow;\ \ Ep \leftarrow C_E(Acc);$$
$$Prog \leftarrow C_P(Acc);\ \ Ip \leftarrow C_0(Acc)$$

would emit code to

- push the environment pointer *Ep*
- push the program pointer *Prog* (in dynamic mode only)
- load *Ep* from the structure pointed to by the accumulator *Acc*
- load *Prog* from the structure pointed to by *Acc* (in dynamic mode only)

- perform a function call (which will save the instruction pointer *Ip*)

The corresponding *return* instruction

$$Fp\uparrow;\ \ c\uparrow;\ \ Ip \leftarrow c_{0;}\ \ Prog \leftarrow c_1;\ \ Ep\uparrow; n\uparrow;\ \ Sp \leftarrow Sp - n - 4$$

would

- pop the frame pointer *Fp* (which was saved by *enter* or *entcol*)
- pop the saved *Ip* into an auxiliary register B_1
- pop *Prog* (in dynamic mode only)
- pop *Ep*
- pop n, the argument count, into an auxiliary register B_2
- adjust the stack pointer *Sp* by B_2 words, removing arguments
- jump to B_1

One might argue that this instruction is complex enough to mandate packaging it in a routine, but then it is interpreted rather infrequently due to tail call optimization, and emission of unnecessary code can be taken care of during code synthesis (see below).

Also note that the above implementation of *apply*/*return* pushes *Prog* as a separate value, so it changes the call frame layout (page 173). The advantage of this approach is that no tuple has to be allocated for saving the return continuation. Its disadvantage is that it makes the stack grow faster (and changes the call frame layout).

In the simple case of the *tailapp* instruction, native "move" instructions can be emitted to move the new argument values to the existing frame of the caller, and the old copies of the new arguments can be removed by adjusting *Sp*.

Instructions like *quote*, *arg*, *ref*, etc translate quite directly to native machine instructions. For each of the instructions an indirect "move" instruction with a displacement (or an equivalent sequence of RISC instructions) is emitted, where the base registers should be allocated globally, if a sufficient number of registers is present. The same applies to *setarg*, *setref*, *cpref*, and *cparg*.

Push, *pop* and related instructions translate directly to machine code, as do the instructions that transfer control locally (*jmp*, etc).

The code of the *enter* instruction is also straight-forward, it is only notable, because it may create a call to a closed subroutine in the case where a wrong number of arguments is detected.

24.3.6 Generating Inline Functions

Inline functions are basically like core instructions, but they do not have inline operands and only process arguments in the *Acc* register and on the stack. Some of them are easily open-coded, some are best implemented as calls to library functions.

It might be worth considering to implement an *unsafe mode* that generates faster code at the cost of type safety and arithmetic overflow checking. In this case, many functions that otherwise require an external routine could be open-coded. For instance, the *times* operation is notoriously hard to implement safely (page 241), but can easily be open-coded in a few instructions on most hardware CPUs.

Implementations of the more complex inline functions can be found in chapter 16, the semantics of simpler functions can be derived from their implementations inside of the LAM in chapter 17.

24.3.7 Code Synthesis

Sequences of stack machine instructions often map to single register machine instructions [Holm1996]. The process of collecting simple instructions in order to create more complex ones if known as *code synthesis*. Here are some sequences of LAM instructions that naturally map to CPU instructions on many platforms.

A predicate (less, equal, etc) followed by a conditional branch instruction maps to a comparison and a conditional branch instruction. The intermediate step of creating a truth value can be omitted. E.g.

```
arg   1
push
quote 2
less
brf   3
```

could translate to

```
MOV  A,FP(1)       Load Acc with first arg.
MOV  B,OBTAB(2)    Load Aux with object #2
CMP  A,B           Compare them
JNL  3             Jump on result not negative
```

This snippet assumes, of course, that stack machine code is being converted to register machine code, as described earlier in this chapter.

The *bit operations* as implemented by the *bitop* instruction (see page 457) can often be open-coded, if the bit operation opcode is known at compile time, which usually will be the case. In particular the AND, OR, XOR, and NOT instructions are typically available on hardware CPUs.

The combination of *tailapp* and *return* is equal to *tailapp*, because the *return* instruction will never execute in this case. Removing the *return* instruction may be considered to be a *peephole optimization*, which examines short sets of instructions and transforms them to different, more efficient sequences.

There are lots of other peephole optimizations that can be performed on stack machine code, like

```
X 0 times  →   0
X 1 times  →   X
X 0 plus   →   X
0 X minus  →   X negate
```

However, occasions for such optimizations often stem from low-level constructs, like array access. In a low-level language, a vector reference like

(VREF *v n***)**

might translate to code that first scales *n* by multiplying it by the

size of a slot and then adds it to the address of the vector V. In the case $n = 0$, this would generate stack machine code like this:

v n **0 times plus ref**

which can be peephole-optimized in two steps:

v **0 plus ref** (*n* **0 times** → **0**)

v **ref** (**0 plus** → **nothing**)

In LISP programs, array access is often done in a subroutine, which also performs *bounds checking*, so this opportunity does not arise as frequently as in more low-level languages.

Of course, even here an unsafe mode may be considered, which open-codes vector access at the cost of omitted bounds checking. Its is left to the compiler writer to decide whether or not such optimizations are worth the additional complexity of the implementation.

APPENDIX

25. Using LISP9

25.1 Building the System

The LISP9 system is distributed in source code form, so it has to be built before it can be used. The core of the system is written in the C programming language as described in [K&R2] and in the documents "ANSI X3.159-1989" of the American National Standards Institute and "ISO/IEC 9899:1990" of the International Standards Organization. The language is widely known as "ANSI C 89" "ISO C 90" or "C89". The language standard of 1989 is sufficient, but compilers implementing later standards should be able to compile the code as well.

There are no external dependencies except for the standard C library, so the following command should compile the core of LISP9 on most Unix systems:

```
cc -o ls9 ls9.c
```

The next step is the compilation of the part of the system that is written in LISP9. This part contains all derived functions and special forms of the system and is sometimes called its "library". In a Unix environment the following commands will compile the library and dump an image file containing the compiled code:

```
rm -f ls9.image
echo '(dump-image "ls9.image")' | ./ls9 -q
```

To make sure that the system is working properly, a number of tests should be run. This can be done automatically by batch-interpreting the test suite. On a Unix system, this can be done by typing:

```
./ls9 test.ls9
```

The test should complete quickly and print "Everything fine!". In case something went wrong, a report can be found in the file "test.log".

There are some additional components of the system that are not contained in the image file compiled above. These optional parts

include:

- the bytecode disassembler
- the grinder (pretty printer)
- the help facility

These components can be included by loading them into the LISP9 system and then updating the image file. The following Unix command would include all of the above:

```
echo "(save)" | \
  ./ls9 -l help.ls9 -l grind.ls9 -l disasm.ls9
```

25.2 Interacting with LISP9

The LISP9 system is started by typing the command

```
./ls9
```

on a Unix system. The system is not intended to be installed, so it always has to be started in the directory that contains the executable file **ls9** and the image file **ls9.image**.

Once started, the system prints a banner and issues a command line prompt (a "*" character):

```
LISP9 20190311
* _
```

LISP expressions typed at the prompt will be evaluated and their results will be printed before the next prompt is issued:

```
* (cons 1 2)
(1 . 2)
* (mapcar list '(a b c) '(1 2 3))
((a 1) (b 2) (c 3))
* _
```

The special variable ** is always bound to the value of the expression most recently evaluated, so that result does not have to be typed in when it is to be used in a subsequent computation:

```
* (mapcar list '(a b c) '(1 2 3))
((a 1) (b 2) (c 3))
* (assq 'b **)
```

```
(b 2)
* _
```

There are some *meta commands* that offer abbreviations for frequently used expressions. All of these commands starts with a comma (",") and they work only at the interpreter prompt. The ",?" command gives a summary:

```
* ,?
,c = syscmd
,h = help
,l = load
nil
* _
```

The ",l" (comma-ell) command loads the specified file. For instance, the meta command

```
,lhelp
```

would expand to the expression

```
(load "help.ls9")
```

The ",h" command allows to browse the LISP9 reference manual, if the file "help.ls9" was previously loaded. Typing just ",h" will explain how to navigate the manual. The command ",h,t" will list the table of contents.

To exit the LISP9 system, either send it an end-of-file marker (by pressing control-D on a Unix system) or type

```
(quit)
```

26. LISP9 Summary

This is a very brief description of LISP9's syntax, special forms and built-in functions. For a more thorough introduction, see the LISP9 Reference Manual, which is contained in the LISP9 distribution archive.

26.1 Definitions

Forms are specified as

form => type

or

(*operator type*1 ···) => *type*

meaning that a form evaluates to an object of a specific type or an operator (function or special operator) applied to some objects of specific types yields a specific type.

The following types exist:

Type	**Description**
a/b	either type *a* or type *b*
alist	association list
char	character
ctag	catch tag
dlist	dotted list (any pair)
eof	end of file marker
expr	any type (as argument)
fixnum	fixnum
fun	function
fun^n	function of *n* arguments
inport	input port
list	proper (**nil**-terminated) list
n	fixnum
obj	any type (as return value)
outport	output port
pair	dotted list (emphasizing the "pair" aspect)

string	string
t/nil	either **t** or **nil**
undefined	no value is returned by a function
unspecific	no specific type, return value should be ignored
vector	vector

26.2 Syntax

; comment => undefined

Comments are *objects* that can be read by **read**! Hence they can be kept intact by code walkers. Unlike in other LISP systems, they may not appear at any place in the code, though. A comment can also be written as **(-- "comment")**.

t => t

The canonical "true" value.

nil => nil

The only "false" value and the end-of-list indicator at the same time. May also be written as **()**.

(fun expr · · ·) => obj

Apply the function *fun* to the given expressions (arguments).

(<special> expr · · ·) => obj

Pass the given expressions to a special operator. The semantics of the individual special operators is given in the following sections.

variable => obj

Variables are created by the **def** and **lambda** special forms (q.v.). A variable evaluates to the value to which it is bound (by **def** or by applying a lambda function to values). The name of a variable is a symbol.

(quote symbol) => symbol

A symbol is a sequence of characters that represents itself.

Symbols can be tested for having the symbol type and for identity (using **eq**). The characters allowed in symbol names are all upper and lower case ASCII letters, decimal digits, and the following special characters:

```
! $ % ^ & * - / _ + = ~ . ? < > :
```

Upper-case characters will be mapped to lower case by **read**.

Symbols must be quoted, otherwise they represent variables.

pairs and lists

A pair is an ordered pair with a car part and a cdr part:

```
(car . cdr)
```

A list is a pair whose car part is any LISP object and whose cdr part is either **nil** or another list, i.e. lists are **nil**-terminated sequences of objects. A list of the form

```
(a . (b . (c . nil)))
```

is usually written as

```
(a b c)
```

Lists must be quoted or they are applications of functions or special operators. Pairs must also be quoted.

fixnum => fixnum

Fixnums are small integer numbers that have various representations, like **123**, **–1**, **+5**, or **#16rfff**.

character => char

Characters are entities that represent printable glyphs or other, non-printable, code points of the ASCII alphabet. They all have a prefix of the form **#** followed by either a glyph, a symbolic name (e.g. **sp**), or an octal code point (e.g. **177**).

string => string

A string is a sequence of ASCII characters enclosed by double quote characters ("). Special characters can be included by prefixing them with a backslash, e.g. **n** is a newline character and

\" is a quote character. Octal sequences can be used to include any character (e.g. **33** = ESC).

vector => vector

Vectors have the same syntax as lists in abbreviated notation (not pair notation) with a **#** character attached. They may contain any type of LISP object:

```
#(foo (list) #(vector) 123)
```

Their elements are accessed in constant time.

26.3 Binding Forms

(def <var> expr) => unspecific

Create the variable *<var>* and store the value of *expr* in the location bound to it.

(defun (<var> . <formals>) expr1 expr2 ...) => unspecific

Short for *(def <var> (lambda <formals> expr1 expr2 ...))*.

(let ((<var1> <arg1>) ...) expr1 expr2 ...) => obj

Create variables *<var>*, then evaluate expressions *<arg>* in no specific order and store their values in the associated variables. With the bindings in effect, evaluate the given expressions. Return the value of the last expression.

(let ((<var> <arg>) ...) expr1 expr2 ...) => obj*

Create the first variable *<var>*, evaluate the associated argument, *<arg>* and store it in the variable. With that binding in effect, create and initialize the next variable. When no variable/argument pairs are left, evaluate the given expressions and return the result of the last expression.

(labels ((<var> <arg>) ...) expr1 expr2 ...) => obj

Initialize variables and evaluate expressions in the same way as **let***. When *<arg>* is a lambda special form, its body may refer to any *<var>*, thereby allowing to create mutually recursive

definitions.

(with ((<var> <arg>) ...) expr1 expr2 ...) => obj

Temporarily change the bindings of the free variables *<var>* to their corresponding *<arg>*s. With these changes in effect, evaluate the given expressions. Return the value of the last expression. Before returning, restore the original values of the variables.

26.4 Functions and Function Application

(lambda <formals> expr1 ... exprN) => fun

Create a function with the variables (formal arguments) *<formals>* and the body *(prog expr1 ... exprN)*. *<Formals>* may be a proper list of variables (fixed arguments), a dotted list, where the last variable binds to a list of extra arguments, or a single variable, which binds to a list of all arguments.

(fun expr1 ...) => obj

Apply the function *fun* to the given expressions: create the variables of the function, then evaluate the *expr*s in no specific order and bind them to the corresponding variables. With the bindings in effect, evaluate the body of the function and return its value.

(apply fun expr1 ... list) => obj

Apply the function *fun* to the arguments in *list*. If there are any additional expressions before *list*, cons them to *list*.

26.5 Conditional Evaluation

(and expr1 ...) => obj

Return the value of the first expression that evaluates to **nil**. When no expression evaluates to **nil**, return the value of the last one, When no expressions are given, return **t**.

(or expr1 ...) => obj

Return the value of the first expression that does not evaluate to **nil**. When all expression evaluate to **nil** or no expressions are given, return **nil.**

(prog expr1 ...) => obj

Evaluate the given expressions from left to right and return the value of the last expression. When no expressions are given, return **nil**.

(if expr1 expr2 expr3) => obj

If *expr1* evaluates to a value other than **nil**, return the value of *expr2*, else return the value of *expr3*.

(if expr1 expr2) => obj

If *expr1* evaluates to a value other than **nil**, return the value of *expr2*, else return an unspecific value.

(if expr1 expr2) => obj*

If *expr1* evaluates to a value other than **nil**, return the value of *expr1*, else return the value of *expr2*.

(cond (<pred> expr ...) ...) => obj

Evaluate clauses from the left to the right. First evaluate the predicate *<pred>* of a clause. When the predicate is true, evaluate the given expressions, if any. Return the value of the last expression. When no expressions are given, return the value of *<pred>*. When no more clauses exist, return an unspecific value.

The last clause may have a predicate of the form **else**, which is always true.

A clause of the form *(pred => fun)* is interpreted as

(let ((G pred)) (if G (fun G)))

where G is a gensym.

(case <key> (list expr1 expr2 ...) ...) => obj

Evaluate *<key>* and test it for membership in each *list* in the subsequent clauses. When a matching clause is found, evaluate the associated expressions and return the value of the last expression. When no clause matches, return an unspecific value.

The last clause may have a predicate of the form **else**, which always matches. Membership is tested using **memv**.

26.6 Loops

(do <bindings> (<test> expr1 ...) <body>) => obj

Each binding has the form *(<var> expr1)* or *(<var> expr1 expr2)*. In either case, variables are created and bound like in the **let** special form, initializing them with *expr1*. When *expr2* is not specified, it defaults to *<var>*.

With *<bindings>* in effect, evaluate *<test>* and if it yields truth, evaluate the associated expressions and return the value of the last expression.

When the test does not succeed, evaluate the expressions in *<body>* and then start over at the point where variables are bound, but this time binding them to *expr2*.

(let <name> ((<var> <arg>) ...) expr1 expr2 ...) => obj

Create the function *<name>* that has the given *<var>*s as arguments and *expr1* ... as body and apply it to the given *<arg>*s. The body of the function may then apply *<name>* in order to recurse.

26.7 Quotation

(quote expr) => obj

Return *expr* as a literal object.

(qquote <template>) => obj

Return the object *<template>*, but replace applications of **unquote**

and **splice** in that object with the values returned by them.

(unquote expr)

Only in **qquote** forms: evaluate and return *expr*.

(splice expr)

Only in **qquote** forms: evaluate *expr*, which must evaluate to a list. Splice the elements of that list into a surrounding **qquote** *<template>*, i.e. remove the application of **splice** and insert a number of individual list elements in its place.

26.8 List Functions

(assoc expr alist) => pair/nil
(assq expr alist) => pair/nil
(assv expr alist) => pair/nil

Find the first pair (association) in the given alist that has *expr* as its key. Return the association. When no association has a matching key, return **nil**. **Assoc** uses **equal** to compare keys, **assv** uses **eqv**, and **assq** uses **eq**.

(atom expr) => t/nil

Return truth, if *expr* is an atom (not a pair).

(caar pair) => obj
...
(cddddr pair) => obj

Return elements of nested pairs, e.g. **caddr** is the car of the cdr of the cdr of its argument.

(car pair) => obj

Return the car part of *pair*.

(cdr pair) => obj

Return the cdr part of *pair*.

(conc list ... expr) => dlist
(nconc list ... dlist) => dlist

Return a fresh list containing the concatenation of the given lists. The last argument may be a non-list or a dotted list, resulting in a dotted list. When no lists are given, return **nil**. **Nconc** is destructive.

(cons expr1 expr2) => pair

Create a fresh pair with *expr1* in its car and *expr2* in its cdr part.

(filter fun^1 *list) => list*

Return a fresh list containing those elements of *list* that satisfy the unary predicate fun^1.

(fold fun^2 *expr list) => obj*
(foldr fun^2 *expr list) => obj*

Fold the binary function fun^2 over the given list using *expr* as the base element. **Fold** folds to the left and **foldr** fold to the right.

(mapcar fun list1 list2 ...) => list
(foreach fun list1 list2 ...) => unspecific

Map the function *fun* over the given lists. Return a fresh list containing the results of applying *fun* to elements of the given list or lists. **Foreach** is like **mapcar**, but maps *fun* over elements of the lists from the left to the right.

(length list) => fixnum

Return the number of elements in the given list.

(list expr ...) => list

Create a fresh list with the given expressions as elements.

(listp expr) => t/nil

Return truth, if *expr* is a proper list, i.e. if it is acyclic and terminated by **nil**.

(member expr list) => list
(memq expr list) => list
(memv expr list) => list

Return the leftmost tail of *list* that has *expr* as its first element. When *list* does not contain *expr*, return **nil**. **Member** uses **equal** to locate elements, **memv** uses **eqv**, and **memq** uses **eq**.

(nth fixnum list) => obj

Return the *fixnum*th element of *list*.

(nth-tail fixnum dlist) => obj

Return the tail of *dlist* that starts with its *fixnum*th element.

(null expr) => t/nil

Return truth, if *expr* is **nil**.

(reconc list expr) => dlist
(nreconc list expr) => dlist

Return a fresh list containing the elements of *list* concatenated in reverse order to *expr*. **Nreconc** is destructive.

(rever list) => list
(nrever list) => list

Return a fresh list containing the elements of *expr* in reverse order. **Nrever** is destructive.

26.9 Mutation

(setq <var> expr) => obj

Evaluate *expr* and store its value in the location bound to the variable *<var>*.

(setcar pair expr) => pair

Destructively change the car part of *pair* to the value of *expr*.

(setcdr pair expr) => pair

Destructively change the cdr part of *pair* to the value of *expr*.

26.10 Equivalence

(eq expr1 expr2) => t/nil

Return truth, if *expr1* and *expr2* are the same object.

(eqv expr1 expr2) => t/nil

Return truth, if *expr1* and *expr2* have the same value (chars having the same code point or fixnums having the same numeric value). **Eqv** implies **eq**.

(equal expr1 expr2) => t/nil

Return truth, if *expr1* and *expr2* are structurally equal, i.e. if they have unique external representations that print the same using **prin**. **Equal** implies **eqv**.

26.11 Type Predicates

(charp expr) => t/nil
(ctagp expr) => t/nil
(inportp x) => t/nil
(fixp x) => t/nil
(funp expr) => t/nil
(outportp x) => t/nil
(pair expr) => t/nil
(stringp expr) => t/nil
(symbolp expr) => t/nil
(vectorp expr) => t/nil

Each type predicate returns truth, if its argument evaluates to an object of the corresponding type.

26.12 Type Conversion

(char fixnum) => char
(charval char) => fixnum

(liststr list) => string
(strlist string) => list
(listvec list) => vector
(veclist vector) => list
(symbol string) => symbol
(symname symbol) => string

The type conversion functions convert an object of one type to an equivalent object of another type. The conversion of type A to type B and then back to type A will yield an object that is structurally equal (**equal**) to the original object.

26.13 Arithmetic Functions

(n1 ...) => fixnum*

Return the product of the given fixnums or 1, if no arguments are given.

(+ n1 ...) => fixnum

Return the sum of the given fixnums or 0, if no arguments are given.

(– n1 n2 ...) => fixnum

Return the difference of the given fixnums or $-n$, if only one argument is given.

(= n1 n2 ...) => t/nil

Return truth, if the given fixnums are equal.

(< n1 n2 ...) => t/nil

Return truth, if the given fixnums are in strict monotonically ascending order.

(> n1 n2 ...) => t/nil

Return truth, if the given fixnums are in strict monotonically descending order.

(<= n1 n2 ...) => t/nil

Return truth, if the given fixnums are in monotonically ascending order.

(>= n1 n2 ...) => t/nil

Return truth, if the given fixnums are in monotonically descending order.

(abs fixnum) => fixnum

Return the magnitude (absolute value) of *fixnum*.

(bitop <op> n1 n2 n3 ...) => fixnum

Apply the bit operation *<op>* to the given fixnum values. See page 457 for a summary of *<op>* values.

(asrb n1 n2 n3 ...) => fixnum
(andb n1 n2 n3 ...) => fixnum
(eqvb n1 n2 n3 ...) => fixnum
(nandb n1 n2 n3 ...) => fixnum
(norb n1 n2 n3 ...) => fixnum
(notb fixnum) => fixnum
(orb n1 n2 n3 ...) => fixnum
(shlb n1 n2 n3 ...) => fixnum
(shrb n1 n2 n3 ...) => fixnum
(xorb n1 n2 n3 ...) => fixnum

Apply bit operations of the given fixnum values. See page 457.

(div n1 n2) => fixnum

Return the truncated quotient of the fixnums *n1* and *n2*.

(mod n1 n2) => fixnum

Return the remainder of the floored division of the fixnums *n1* and *n2*.

(rem n1 n2) => fixnum

Return the remainder of the truncated division of the fixnums *n1* and *n2*.

(evenp fixnum) => t/nil
(oddp fixnum) => t/nil

Return truth, if *fixnum* is even or odd, respectively.

(expt n1 n2) => fixnum

Return the fixnum *n1* raised to the power of *n2*.

(gcd n1 n2) => fixnum

Return the greatest common divisor of the fixnums *n1* and *n2*

(lcm n1 n2) => fixnum

Return the least common multiple of the fixnums *n1* and *n2*

(max n1 n2 ...) => fixnum

Return the largest of the given fixnums.

(min n1 n2 ...) => fixnum

Return the least of the given fixnums.

(not expr) => t/nil

Return the logical complement of *expr*.

26.14 Character Functions

(alphac char) => t/nil
(lowerc char) => t/nil
(numeric char) => t/nil
(upperc char) => t/nil
(whitec char) => t/nil

Return truth, if the given char object is alphabetic, lower-case, numeric, upper-case, or white space, respectively.

(c= c1 c2 ...) => t/nil

Return truth, if all the given characters share the same code point.

(c< c1 c2 ...) => t/nil
(c> c1 c2 ...) => t/nil

(c<= c1 c2 ...) => t/nil
(c>= c1 c2 ...) => t/nil

Return truth, if the code points of all given characters are in the order imposed by the corresponding predicate. (See **<**, **>**, etc.)

(downcase char) => char

If *char* is an upper-case letter, return the lower-case variant of that letter, else return *char*.

(upcase char) => char

If *char* is a lower-case letter, return the upper-case variant of that letter, else return *char*.

26.15 String Functions

(mkstr fixnum char) => string
(mkstr fixnum) => string

Return a fresh string of the given length, filled with *char*. When no char is given, it defaults to the blank character.

(numstr n1) => string
(numstr n1 n2) => string

Return a fresh string containing the readable representation of the fixnum *n1* to the base *n2*. When no base is given, it defaults to (decimal) 10.

(s= s1 s2) => t/nil

Return truth, if the string *s1* contains the same characters, in the same order, as *s2*.

(s< s1 s2) => t/nil
(s<= s1 s2) => t/nil
(s> s1 s2) => t/nil
(s>= s1 s2) => t/nil

Return truth, if the strings *s1* and *s2* are in the lexicographic order imposed by the corresponding predicate. (See **<**, **>**, etc). The

comparison is sensitive to case.

(sconc string ...) => string

Return a fresh string containing the concatenation of the given strings. When no strings are given, return an empty string.

(scopy string) => string

Return a fresh copy of the given string.

(sfill string char) => string

Fill the given string with *char*. This is a destructive operation.

(si= s1 s2) => t/nil

Return truth, if the string *s1* contains the same characters, in the same order, as *s2*. Upper-case and lower-case variants of the same character are considered to be equal.

(si< s1 s2) => t/nil
(si<= s1 s2) => t/nil
(si> s1 s2) => t/nil
(si>= s1 s2) => t/nil

Return truth, if the strings *s1* and *s2* are in the lexicographic order imposed by the corresponding predicate. (See **<**, **>**, etc). The comparison ignores the case of any letters in the strings.

(sref string fixnum) => char

Return the *fixnum*th character of *string*.

(sset string fixnum char) => string

Replace the *fixnum*th character of *string* with *char*. Return an unspecific value. This is a destructive operation.

(ssize string) => fixnum

Return the number of characters in the given string.

(string char ...) => string

Return a fresh string containing the given characters.

(strnum string) => fixnum/nil
(strnum string fixnum) => fixnum/nil

Parse the readable representation of an integer number to the base of *fixnum* in *string* and return its numerical value. When no base is given, it defaults to (decimal) 10. When the string does not contain the representation of a valid number or if an overflow occurs while parsing the number, return **nil**.

(substr string n1 n2) => string

Return a fresh string containing the characters from position *n1* up to, but *not* including, position *n2* in *string*.

26.16 Vector Functions

(mkvec fixnum) => vector
(mkvec fixnum expr) => vector

Return a fresh vector of the given length, filled with *expr*. When *expr* is not given, it defaults to **nil**.

(subvec vector n1 n2) => vector

Return a fresh vector containing the elements from position *n1* up to, but *not* including, position *n2* in the given vector.

(vconc vector ...) => vector

Return a fresh vector containing the concatenation of the given vectors. When no arguments are given, return an empty vector.

(vector expr ...) => vector

Return a fresh vector consisting of the given elements.

(vfill vector expr) => vector

Fill the given vector with *expr*. This is a destructive operation.

(vref vector fixnum) => obj

Return the $fixnum^{th}$ element of *vector*.

(vset vector fixnum expr) => vector

Replace the *fixnum*th element of *vector* with *expr*. Return an unspecific value. This is a destructive operation.

(vsize vector) => fixnum

Return the number of elements in the given vector.

26.17 Macros

(macro <keyword> fun) => unspecific

Evaluate the function *fun* and bind it to the name *<keyword>* in the macro environment. *<Keyword>* becomes the keyword of a new derived special form.

(defmac (<kw> . <formals>) expr1 expr2 ...) => unspecific

Short for *(macro <kw> (lambda <formals> expr1 expr2 ...))*.

(gensym) => symbol

Return an uninterned (almost) unique symbol.

(mx expr) => obj

Fully macro-expand the given expression.

(mx1 expr) => obj

Perform a single macro-expansion step on the given expression.

26.18 Non-Local Exits

(catch fun^1*) => obj*
(catch fun*^{1}*) => obj*

Establish a catch tag (a non-local exit point) and pass it to the function *fun*^{1}. The function can use the catch tag to return to the point where **catch** returns. **Catch** registers on the unwind stack, but **catch*** does not.

(throw ctag expr) => undefined
(throw ctag expr) => undefined*

Pass control to the point marked by the catch tag *ctag*. Evaluation will continue at the point where the catch form (**catch** or **catch***) that established *ctag* returns. The catch form will return the value of *expr*. **Throw** will trigger the evaluation of **unwind** forms, but **throw*** will not.

(unwind fun^0_1 fun^0_2*) => obj*

Evaluate the thunk (nullary function) fun^0_2 and then evaluate fun^0_1. Return the value returned by fun^0_2. The thunk fun^0_1 will also evaluate when control passes from a **throw** form inside of fun^0_2 to a **catch** tag outside of the **unwind** form.

(error string expr) => undefined
(error string) => undefined

Report an error and abort evaluation. When *expr* is specified, it will be used to indicate the cause of the error.

(catch-errors (<errval>) expr1 expr2 ...) => obj
(catch-errors () expr1 expr2 ...) => obj

Catch all error conditions that occur in the given expressions and return *<errval>* instead of signaling an error. No further expressions in the body of **catch-errors** will evaluate after catching an error. When no error is caught, the value of the last expression is returned. When no *<errval>* is specified, a string containing an error message will be returned.

26.19 Input/Output Functions

(close-port port) => unspecific

Close the given input or output port.

(delete string) => unspecific

Delete the file specified in *string*.

(errport) => outport

(inport) => inport
(outport) => outport

Return the current error, input, or output port, respectively.

(existsp string) => t/nil

Return truth, if the file specified in *string* exists.

(eofp expr) => t/nil

Return truth, if *expr* is an EOF marker.

(open-infile string) => inport

Open the file specified in *string* for input, connect it to an input port and return that port.

(open-outfile string) => outport
(open-outfile string t) => outport

Open the file specified in *string* for output, connect it to an output port and return that port. When a second argument (usually **t**) is given, append output, if the file already exists. When only one argument is given, truncate any existing file of the same name to zero length.

(format expr) => string

Write the external representation of *expr* to a fresh string and return that string.

(prin expr) => obj
(prin expr outport) => obj

Write the external representation of *expr* to the given output port. When no port is given, write to the current output port. Return the expression.

(princ expr) => obj
(princ expr outport) => obj

Write a human-readable (but not necessarily unambiguous) representation of *expr* to the given output port. When no port is

given, write to the current output port. Return the expression.

(print expr ...) => unspecific

Write the external representations of the given expressions, separated by blanks, to the current output port. Then write a newline sequence. Return an unspecific value.

(read) => obj/eof
(read inport) => obj/eof

Read the external representation of a LISP object from the given input port and return the object. When no port is given, read from the current input port. When the EOF is reached before reading an object, return an EOF marker.

(read string) => list/string

Read the external representation of a LISP object from the given string and return that object in a singleton list. When an error occurs during reading, return a message describing the error in a string.

(readc) => char/eof
(readc inport) => char/eof
(peekc) => char/eof
(peekc inport) => char/eof

Read a single character from the given input port and return it. When no port is given, read from the current input port. **Peekc** reads a character without moving the input pointer, i.e. a subsequent **peekc** or **readc** will yield the same character. When no characters are available from the given port, return an EOF marker.

(readln) => string/eof
(readln inport) => string/eof

Read characters from the given port until a newline character or the EOF is reached. Place the received characters (excluding any newline character) in a string and return it. When no characters are received before the EOF is reached, return an EOF marker.

(set-inport inport) => unspecific
(set-outport outport) => unspecific

Set the current input or output port to the specified port. This is is a destructive operation.

(terpri) => #\nl
(terpri outport) => #\nl

Write a newline sequence to the given output port. When no port is specified, write to the current output port.

(with-infile string fun^0*) => obj*
(with-outfile string fun^0*) => obj*

Redirect the current input or output port to the file specified in *string*. With the redirection in effect, evaluate the thunk (nullary function) fun^0. Then close the file opened for redirection, reset the current input or output port to the port previously in effect and return the value returned by fun^0.

(with-inport string fun^1*) => obj*
(with-outport string fun^1*) => obj*

Open the file given in *string* for input or output and pass the resulting port to the function fun^1. When the function returns, close the port and return the value returned by fun^1.

(writec char) => char
(writec char outport) => char

Write the character *char* to the given output port. When no port is specified, write to the current output port. Return the character written.

26.20 System Functions

(cmdline) => list

Return a list of command line arguments passed to a program.

(constp expr) => t/nil

Return truth, if the value of *expr* is an immutable object.

(dump-image string) => unspecific

Write the current state of the system to the image file specified in *string*.

(save) => unspecific

Write the current state of the system to the image file from which it has been loaded at startup time.

(eval expr) => obj

Evaluate *expr* and returns its value.

(gc) => list

Perform a garbage collection and return the total number of free nodes and vector cells in a list.

(load string) => unspecific

Load (read and evaluate) expressions from the given file.

(obtab) => vector

Return a vector containing all literal data objects currently known to the system.

(start) => unspecific

The body of this (user-defined) function will evaluate when the system starts up.

(symtab) => vector

Return a vector containing all interned symbols that are currently known to the system.

(syscmd string) => fixnum

Pass *string* to the operating system for execution.

(untag expr) => obj

Remove the tag from a tagged data object. Do not use this function!

27. LISP9 Compared to Scheme

Many LISP9 programs translate pretty directly to Scheme [R4RS] and vice versa. Below follows a list of differences between the languages.

Canonical truth is **t** in LISP9 and **#t** in Scheme, falsity is **nil** in LISP9 and **#f** in Scheme. **nil** is **'()** in Scheme and is a "true" value in most R4RS and all post-R4RS implementations.

Scheme implements a variety of numeric types, from integer to complex, while LISP9 only provides small (fixnum) integers.

Scheme is the bigger of the two languages. The below list indicates many LISP9 functions that do not have any counterpart in Scheme, but there are at least as many Scheme functions that have no counterpart in LISP9. Many LISP9 functions can probably be replaced by functions defined in some Scheme Request For Implementation [SRFI].

The following functions and special forms may be used interchangeably in many cases:

LISP9	**Scheme**
*	*
+	+
-	-
--	*n/a*
<	<
<=	<=
=	=
>	>
>=	>=
abs	**abs**
alphac	**char-alphabetic?**
and	**and**
andb	*n/a*
apply	**apply**
asrb	*n/a*

LISP9	Scheme
assq	assq
assv	assv
assoc	assoc
atom	(lambda (x) (not (pair? x)))
bitop	*n/a*
c<	char<?
c<=	char<=?
c=	char=?
c>	char>?
c>=	char>=?
caar...cddddr	caar...cddddr
car	car
case	case
catch	*n/a*
catch*	call-with-current-continuation (1)
catch-errors	*n/a*
cdr	cdr
char	integer->char
charp	char?
charval	char->integer
close-port	close-input-port, close-output-port
cmdline	*n/a*
conc	append
cond	cond
cons	cons
constp	*n/a*
ctagp	*n/a* (2)
def	define
defmac	*n/a* (3)
defun	define
delete	*n/a*
div	quotient
do	do
downcase	char-downcase
dump-image	*n/a*

LISP9	Scheme
`eofp`	`eof-object?`
`eq`	`eq?`
`eqv`	`eqv?`
`eqvb`	*n/a*
`equal`	`equal?`
`error`	*n/a*
`errport`	*n/a*
`eval`	*n/a*
`evenp`	`even?`
`existsp`	*n/a*
`expt`	`expt`
`filter`	*n/a*
`fixp`	`integer?`
`fold`	*n/a*
`foldr`	*n/a*
`foreach`	`for-each`
`format`	*n/a*
`funp`	`procedure?`
`gc`	*n/a*
`gcd`	`gcd`
`gensym`	*n/a* (3)
`if`	`if`
`if*`	*n/a*
`inport`	`current-input-port`
`inportp`	`input-port?`
`labels`	`letrec` (4)
`lambda`	`lambda`
`lcm`	`lcm`
`length`	`length`
`let`	`let` (5)
`let*`	`let*`
`list`	`list`
`listp`	`list?`
`liststr`	`list->string`
`listvec`	`list->vector`

LISP9	Scheme
`load`	`load`
`lowerc`	`char-lower-case?`
`macro`	*n/a* `(3)`
`mapcar`	`map`
`max`	`max`
`member`	`member`
`memq`	`memq`
`memv`	`memv`
`min`	`min`
`mkstr`	`make-string`
`mkvec`	`make-vector`
`mod`	`modulo`
`mx`	*n/a*
`mx1`	*n/a*
`nandb`	*n/a*
`nconc`	*n/a*
`norb`	*n/a*
`not`	`not`
`notb`	*n/a*
`nreconc`	*n/a*
`nrever`	*n/a*
`nth`	`list-ref`
`nth-tail`	`list-tail`
`null`	`null?`
`numeric`	`char-numeric?`
`numstr`	`number->string`
`obtab`	*n/a*
`oddp`	`odd?`
`open-infile`	`open-input-file`
`open-outfile`	`open-output-file`
`or`	`or`
`orb`	*n/a*
`outport`	`current-output-port`
`outportp`	`output-port?`
`pair`	`pair?`

LISP9	Scheme
`peekc`	`peek-char`
`prin`	`write`
`princ`	`display`
`print`	*n/a*
`prog`	`begin`
`qquote`	`quasiquote`
`quit`	*n/a*
`quote`	`quote`
`read`	`read`
`readc`	`read-char`
`readln`	*n/a*
`reconc`	*n/a*
`rem`	`remainder`
`rever`	`reverse`
`s<`	`string<?`
`s<=`	`string<=?`
`s=`	`string=?`
`s>`	`string>?`
`s>=`	`string>=?`
`save`	*n/a*
`sconc`	`string-append`
`scopy`	`string-copy`
`set-inport`	*n/a*
`set-outport`	*n/a*
`setcar`	`set-car!`
`setcdr`	`set-cdr!`
`setq`	`set!`
`sfill`	`string-fill!`
`shlb`	*n/a*
`shrb`	*n/a*
`si<`	`string-ci<?`
`si<=`	`string-ci<=?`
`si=`	`string-ci=?`
`si>`	`string-ci>?`
`si>=`	`string-ci>=?`

LISP9	**Scheme**
splice	**unquote-splicing**
sref	**string-ref**
sset	**string-set!**
ssize	**string-length**
start	*n/a*
string	**string**
stringp	**string?**
strlist	**string->list**
strnum	**string->number**
substr	**substring**
subvec	*n/a*
symbol	**string->symbol**
symbolp	**symbol?**
symname	**symbol->string**
symtab	*n/a*
syscmd	*n/a*
terpri	**newline**
throw	*n/a*
throw*	**call-with-current-continuation (1)**
unquote	**unquote**
untag	*n/a*
unwind	*n/a*
upcase	**char-upcase**
upperc	**char-upper-case?**
vconc	*n/a*
veclist	**vector->list**
vector	**vector**
vectorp	**vector?**
vfill	**vector-fill!**
vref	**vector-ref**
vset	**vector-set!**
vsize	**vector-length**
with	*n/a*
whitec	**char-whitespace?**
with-infile	**with-input-from-file**

LISP9	Scheme
`with-inport`	`call-with-input-port`
`with-outfile`	`with-output-to-file`
`with-outport`	`call-with-output-port`
`writec`	`write-char`
`xorb`	*n/a*

(1) **Call-with-current-continuation** can generally be used to implement **catch*** and **throw***, but not vice-versa. **Catch** and **throw** have no counterparts in Scheme, but their code can probably be translated to Scheme without much difficulty.

(2) Scheme does not distinguish between procedures and reified continuations, so **procedure?** cannot be used to differentiate between catch tags/continuations and procedures. It can be used to identify contintutations when the difference does not matter, though.

(3) Some Scheme systems provide a low-level macro facility, like **define-macro**, but it is not specified in any Scheme standard. There is a low-level macro facility described in [R4RS], but it does not seem to be used widely.

(4) For differences between **labels** and **letrec**, see page 355. The short version is that **labels** implements what is widely known as **letrec*** rather than **letrec**.

(5) The LISP9 implementation of **let** does include named **let**.

28. BITOP Operations

The following table describes the operations carried out when evaluating the expression **(bitop** *Op A B***)**.

Op	Operands A:0 B:0	 A:0 B:1	 A:1 B:0	 A:1 B:1	**Description**
0	0	0	0	0	0; A, B ignored
1	0	0	0	1	(andb A B)
2	0	0	1	0	(andb A (notb B))
3	0	0	1	1	A
4	0	1	0	0	(andb (notb A) B)
5	0	1	0	1	B
6	0	1	1	0	(xorb A B)
7	0	1	1	1	(orb A B)
8	1	0	0	0	(notb (orb A B))
9	1	0	0	1	(notb (xorb A B))
10	1	0	1	0	(notb B); A ignored
11	1	0	1	1	(orb A (notb B))
12	1	1	0	0	(notb A); B ignored
13	1	1	0	1	(orb (notb A) B)
14	1	1	1	0	(notb (andb A B))
15	1	1	1	1	1; A, B ignored
16					A shifted left by B bits
17					A shifted right by B bits
18					arithmetic right shift

Fig. 59 – Bit operations

29. Bibliography

[AIM116A]
(No author given); AI Memo No. 116A; “PDP-6 LISP (LISP 1.6) Revised”; Massachusetts Institute of Technology, 1967

[AIM349]
Gerald Jay Sussman, Guy Lewis Steele, Jr.; “SCHEME - an Interpreter for Extended Lambda Calculus”; AI Memo No. 349; Massachusetts Institute of Technology, AI Laboratory, 1975

[AIM848]
William Clinger (editor); “The Revised Revised Report on Scheme, an uncommon Lisp”; AI Memo No. 848, Massachusetts Institute of Technology, AI Laboratory, 1985

[Appel1992]
Andrew W. Appel; “Compiling with Continuations”; Cambridge University Press, 1992

[Baker1978]
Henry G. Baker, Jr.; “List Processing in Real Time on a Serial Computer”; Communications of the ACM 21, 4 (April 1978), 280-294

[Chicken]
http://www.call-cc.org/

[Church1941]
Alonzo Church; “The Calculi of Lambda Conversion”; Princeton University Press, 1941

[CHO1988]
William D. Clinger, Anne Hartheimer, Eric Ost; “Implementation strategies for continuations”; Proceedings of the 1988 ACM conference on LISP and functional programming - LFP ’88. pp. 124-131

[CR1991]
William Clinger, Johnathan Rees; “Macros that work”; POPL 1991, Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Pages 155-162

[CLtL1984]
Guy L. Steele, Jr, et. al.; "COMMON LISP The Language"; Digital Press, 1984

[DB1964]
L. Peter Deutsch, Edmund C. Berkeley; "The LISP Implementation of the PDP-1 Computer"; DECUS Program Library, No 85, March 1964

[DHB1992]
R. Kent Dybvig, Robert Hieb, and Carl Bruggeman; "Syntactic abstraction in Scheme"; Lisp and Symbolic Computation 5, 4, pp. 83-110, 1992

[Gabriel1986]
Richard P. Gabriel; "Performance and Evaluation of Lisp Systems"; MIT Press, 1986

[Holm1996]
Nils M Holm; "Lightweight Compiler Techniques"; Self-published, 1996; Lulu Press, 2002

[Holm2014]
Nils M Holm; "Scheme 9 from Empty Space – A Guide to Implementing Scheme in C"; Lulu Press 2014

[Holm2016]
Nils M Holm; "Compiling Lambda Calculus"; Lulu Press 2016

[HOPL1993]
Richard P. Gabriel, Guy L. Steele, Jr.; "The Evolution of Lisp" in Proceedings, HOPL-II, The second ACM SIGPLAN conference on history of programming languages; ACM Press, 1993

[Johnsson1985]
Thomas Johnsson; "Lambda lifting: transforming programs to recursive equations" Proceedings 1985 Conference on Functional Programming Languages and Computer Architectures, Lecture Notes in Computer Science 201, Nancy, France; Springer 1985

[JL1996]
Richard Jones, Rafael D. Lins; "Garbage Collection: Algorithms for Automatic Dynamic Memory Management"; Wiley & Sons, 1996

[K&R2]
Brian W. Kernighan, Dennis M. Ritchie; "The C Programming Language, Second Edition"; Prentice Hall, 1988

[Knuth1968]
Donald E. Knuth; “The Art of Computer Programming, Vol. I: Fundamental Algorithms”; Addison-Wesley, 1968

[Knuth1969]
Donald E. Knuth; “The Art of Computer Programming, Vol. II: Seminumerical Algorithms”; Addison-Wesley, 1969

[Kogge1991]
Peter M. Kogge; “The Architecture of Symbolic Computers”; McGraw-Hill, 1991

[Koopman1994]
Philip J. Koopman, Jr; “A Preliminary Exploration of Optimized Stack Code Generation”; Journal of Forth Applications and Research, 1994, 6(3) pp. 241-251

[Kurlansky2006]
Mark Kurlansky; “Nonviolence, the History of a Dangerous Idea”; Modern Library Chronicles, 2006

[Landin1964]
Peter J. Landin; “The Mechanical Evaluation of Expressions”; The Computer Journal 6 (4): 308-320, 1964

[Landin1965]
Peter J. Landin; “A Generalization of Jumps and Labels”; UNIVAC Systems Programming Research Technical Report, 1965; Reprinted in Higher-Order and Symbolic Computation. 11 (2): 125-143, December 1998.

[McCarthy1960a]
John McCarthy; “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”; Communications of the ACM (CACM) Volume 3 Issue 4, April 1960, Pages 184-195

[McCarthy1960b]
John McCarthy, et. al.; “LISP I Programmer’s Manual”; Computation Center and Research Laboratory of Electronics, M.I.T., 1960

[Milner1978]
Robin Milner; “A Theory of Type Polymorphism in Programming”; Journal of Computer and System Sciences, 1978, Vol 18, Pages 348-375

[MIT-Scheme]
https://www.gnu.org/software/mit-scheme/

[Moon1974]
David A. Moon; "MACLISP Reference Manual"; Massachussetts Institute of Technology, 1974

[Moon1984]
David A. Moon; "Garbage Collection in a Large LISP System"; in ACM Symposium on LISP and Functional Programming, 235-246; Austin, Texas, 1984

[Plotkin1981/2004]
Gordon D. Plotkin; "A Structural Approach to Operational Semantics" Tech. Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1991; With corrections: Journal of Logic and Algebraic Programming 60-61:17-139, July 2004

[R4RS]
William Clinger, Jonathan Rees (editors); "The Revised(4) Report on the Algorithmic Language Scheme"; ACM SIGPLAN Lisp Pointers Volume IV Issue 3, July, 1991, Pages 1-55

[S9fES]
https://t3x.org/s9fes/

[SB2006]
Mark Shannon, Chris Bailey; "Global Stack Allocation: Register Allocation for Stack Machines"; Proceedings, EuroForth 2006

[Smith1996]
Einar Smith; "Elementare Berechenbarkeitstheorie"; Springer, 1996

[Spengler2003]
Brad Spengler; "PaX: The Guaranteed End of Arbitrary Code Execution"; grsecurity.net, 2003

[SRFI]
"Scheme Requests For Implementation"; http://srfi.schemers.org/

[SICP1985]
Harold Abelson, Gerald Jay Sussman, Julie Sussman; "The Structure and Interpretation of Computer Programs"; MIT Press, 1985

[SW1967]
Herbert Schorr, William M. Waite; "An efficient machine-independent procedure for garbage collection in various list structures"; Communications of the ACM (CACM), Volume 10 Issue 8, Aug. 1967, 501-506

30. List of Figures

1	Node pool layout	18
2	Tagged data object layout	22
3	Ring buffer	30
4	Object table	44
5	Live nodes in the heap structure	47
6	GC state M0,T0 (unmarked)	49
7	GC state M1,T1 (marked, car traversal in progress)	50
8	GC state M1,T0 (marked, car traversal complete)	50
9	GC state M1,T0 (marked, final)	51
10	Pointer reversal in the DSW algorithm	51
11	Use of the index field in vector traversal	52
12	Live objects on the runtime stack	56
13	Node allocation	59
14	Vector pool fragmentation	61
15	Vector pool structure	61
16	Vector pool compaction	63
17	Vector object layout	70
18	Hash table structure	73
19	Hash value computation	75
20	Symbol table structure	79
21	External representation examples	92
22	Predefined symbols and their variables (exceptions)	93
23	The spine of the list (a b c), shown in grey	99
24	Meta commands (abbreviations)	108
25	Abbreviated quotation	111
26	The LISP9 compiler	127
27	List structure as abstract syntax tree	128
28	Application and concatenation	129
29	Propagation of bindings	143
30	Finding free variables, examples	146
31	Local variables of APPCONV	161
32	Abstraction layers of the LISP9 abstract machine	168
33	The components of the LAM	170
34	Stack frame layout	173
35	Tail application, simple case ($n = m$)	178

36 Tail application, complex case ($n > m$) 179
37 Building an argument list 180
38 Collecting optional arguments 181
39 Collecting optional arguments, special case 182
40 Inline function application with 3 arguments 186
41 Abstract machine code with inlined data objects 188
42 Bytecode with data objects in an object table 188
43 Bytecode container 193
44 Types of subroutines 197
45 Backpatching a forward jump 205
46 Jump around jump 206
47 The process of macro expansion, two models 228
48 Image file format 273
49 The structure of a closure object 286
50 Non-local exit 292
51 The structure of a catch tag object 292
52 Two environments sharing the same bindings 320
53 Tortoise and hare algorithm 370
54 Modulo and remainder 377
55 Unwinding the unwind stack 383
56 BOYER benchmark results 406
57 Bytecode in a dynamic environment 412
58 Bytecode in a stand-alone environment 412
59 Bit operations 457

31. Index

%arg 159
%closure 153
%compare 393
%ref 159, 159
** 319, 325, 424
errval 294
imagefile 278, 319
quiet 276
unwind 381
– 319, 424
–– 122
< 243
<= 319
= 243, 393
> 243
>= 319

—A–Z—
ASLR 271
AST 43, 128
ATOM_TAG 19
Acc 171, 282
Argv 295, 337
Blank 214
BOYER 406
Bytecode 186
Car 19, 42
Cdr 19, 42
CHUNKSIZE 15, 81, 190, 193, 324
CONST_TAG 20, 99
CPU 168
CTS 195
Cts 195
Deutsch-Schorr-Waite 48
DOT 18
DSW algorithm 48
E0 171, 282
Emitbuf 192, 224
Env 151
Envp 151
EOFMARK 17, 25, 111
Ep 171, 282
Errport 36, 88
Errtag 33
Euclid's algorithm 376
Files 31
Fp 171, 282
Freelist 20, 56, 58
Freevec 20, 64
GC 41
GC protection 66, 88
GC roots 56
GC thrashing 59
GC_roots 56, 334
GC_verbose 56
Glob 33, 90, 162, 319
Handler 33
Here 193, 224
I/O ports 18, 36, 71, 115
I_a 149
I_arg 159
I_closure 153
I_e 149
I_ref 159
IMAGEFILE 14
IMAGESRC 14
Imagevars 273, 334
Inlist 95
Inport 36, 88
Instr 36
Intr 95, 123, 323
Ip 171, 282

IOF 140
ISIZE0 45
ISIZE1 45
ISIZE2 45
LAM 168, 282, 413
Line 31
LISP-1 408
LISP-2 408
LISP-N 408
LISP9 Abstract Machine 168, 282, 413
LOCK_TAG 20, 56
LP 96
LSUBR0 216
LSUBR1 219
MACLISP 110
Macros 229, 236
MARK_TAG 20, 49, 56
Member 374
Memv 374
MXMAX 15, 235
Mxlev 235, 323
NIL 17
NNODES 14
NPORTS 14
NTRACE 16
Nullstr 69
Nullvec 70
NVCELLS 14
OBALLOC 44
Obarray 43, 189
OBFREE 44
Obhash 189
Obmap 43, 189
Obptr 189
OBTAB 43, 56, 191
OBUSED 44, 56
Oddp 376
One 214
OP_QUOTE 45
OSUBR0 213
OSUBR1 214
Outmax 36
Outport 36, 88
Outptr 36
Outstr 36
P_... 94
Plimit 31, 125
PORT_TAG 20
Port_flags 36
Ports 36, 56, 327
PRDEPTH 16, 123
Prog 171, 282
Protected 66
Quiet 324
Quoting 95
RAW_VECDATA 62
RAW_VECLINK 62
RAW_VECSIZE 62
RPAREN 18
Readerr 97
Rejected 36
REPL 33, 33
Restart 33, 338
RP 96
RTS 172
Rts 56, 282, 324, 333
Run 295, 323
S-expression 145
S_... 93
S_errtag 33
S_errval 33
SECD machine 169
SIGINT 95
Sp 56, 171, 282
SUBR 197
Symbols 31, 80, 184
Symhash 80
Symptr 80
SYMTAB 79

Sz 282
T_BYTECODE 23, 192
T_CATCHTAG 23
T_CHAR 23
T_CLOSURE 23
T_FIXNUM 23
T_INPORT 23
T_INPORT 71
T_OUTPORT 23, 71
T_STRING 23
T_SYMBOL 23
T_VECTOR 23
Tag 19, 42
Ten 214
TLE 170
Tmp 66
Tmp_car 59
Tmp_cdr 59
TOKLEN 14
Tp 30
Trace 30, 297
TRAV_TAG 20, 49
TRUE 17
UNDEF 17, 31, 77, 166
USED_TAG 20, 52, 56
VECTOR_TAG 19
Vectors 19, 42
Zero 214

—A—

abbreviation 108
abstract machine code 127
abstract syntax tree 43, 128
accessor 24
accumulator 171
add() 240
add_ovfl() 68
address space layout randomization 271
alloc_nodepool() 42
alloc_vecpool() 42
allocator 20
and 349
andb 375
appconv() 160, 197
append mode 258
applis instruction 180
applis() 288
applist instruction 180
apply 131, 209, 396
apply instruction 177
apply() 286
arg instruction 175
arg() 284
argbox() 284
argref() 284
argument 284, 288
- fixed 180, 288
- formal 149
- optional 180

argvec() 337
asrb 375
assoc 73, 374
association list 72, 90
assq 374
assq() 33, 90
assv 374
atom 19, 20, 26, 59, 68, 121, 136, 221
atomp() 26

—B—

b_ prefix 250
b_gc() 278
b_mkstr() 250
b_mkvec() 254
b_prin() 260
b_read() 259
b_readc() 259
b_symbol() 263

backpatching 206
base element 365
batch compilation 410
begin_rec() 321
benchmark 406
binary inline function 212
binding 138, 183, 352
  deep 90
  local 139, 279
  macro 184
  shallow 90, 141
bindset() 33, 91
bit operation 419
bit operator 218
bitop 218, 390
bitop() 244
blockwrite() 38
body 236, 364
bootstrapping 347
bound variable 138, 149
bounds checking 420
box 90, 138, 176, 283
box() 283
boxing 180
boxref() 283
boxset() 283
brf instruction 176
brt instruction 176
built-in function 197
byte 17
bytecode 23, 52, 281
bytecode buffer 15
bytecode container 192, 224, 282
bytecode generation 403

—C—

c....r 371
c...r 346
c< 245
c<= 245
c= 245
c> 245
c>= 245
caar 346
caar() 21
cadr 346
canonical external form 115
canonical external representation 93
car 346
car field 18
car() 21
carof() 165
case 358
catch 381
catch* 211
catch instruction 183
catch tag 33, 115, 183, 184, 293
catch-errors 294, 296, 384
cconv() 163
cdar 346
cddddr 371
cddddr() 21
cdddr 346
cddr 346
cdr 346
cdr field 18
cdr() 21
cell 17
cequal() 245
cgrtr() 245
cgteq() 245
char 69, 98, 245
char name 97
  octal 96
char object 116
character 18, 69
  white space 246
character literal 14
charp() 25
charval() 69

chunk 15, 45
ckapply() 131
ckargs() 130
ckcomment() 131
ckdef() 132
ckif() 132
ckifstar() 132
cklambda() 134
ckmacro() 135
ckprog() 135
ckquote() 136
ckseq() 131
cksetq() 136
clear() 283, 296
cless() 245
close_port() 87
closure 138, 148, 172, 182, 191, 210, 285
closure conversion 138, 402
closure instruction 182
closure object 411
closure() 286
closure_env() 286
closure_ip() 286
closure_prog() 286
closurep() 25
clrtrace() 30
clsconv() 167, 207
clsobj 286
clteq() 245
cmdarg() 336
code size 407
code synthesis 418
combinator 144, 183
comment 93, 122, 131
comment() 109
commentp() 122
compaction 61
compapp() 210
compapply() 209
compare 394
comparison predicate 219
compbitop() 218
compcls() 207
compexpr() 221
compif() 205
compilation 401
compile time stack 195, 206
compile() 224
compiler 127
complsubr0() 216
complsubr1() 219
composubr0() 213
composubr1() 214
compprog() 203
compsetq() 204
compsubr0() 211
compsubr1() 211
compsubr2() 212
compsubr3() 212
conc 261
conc() 84
cond 348, 357
cons 346
cons cell 14, 18, 26, 288
cons() 23
cons3() 23, 58
conses() 288
constant 20
constp() 26
constructor 23
contains() 157
context 183, 291
  current 293
  tail application 203
continuation 169
convention (list) 26
cparg instruction 183
cpopval() 195
cpref instruction 183

cpushval() 195
ctagp() 25
current context 293
current continuation 293
current input port 379
current output port 379
cyclic cons structure 369

—D—

data object 42, 92
deep binding 90
def 132, 142, 162, 232
defconv() 162
definition 132,
defmac 135, 234
defun 232, 234
depth 123
derived special form 226
disassembly 404
division remainder 377
do 363
dotted list 26, 99, 119
dottedp() 207
dump-image 278
dump_image() 278
dumpimg() 273
dynamic extent 184, 361
dynamic scope 362

—E—

earmuff 363
efficiency 170
emit buffer 224
emit() 193
emitarg() 194
emitop() 194
emitq() 194
emitter 193
empty list 99
end of file 111
end_rec() 321
entcol instruction 180
entcol() 289
enter instruction 177
entry point 173
envbox() 284
environment 90, 145
  global 33, 90
  local 139, 154, 172, 207
  top level 142, 151, 170, 319
environment pointer 171
eof marker 115
eofp() 25
eq 372
equal 373
equal() 243
equality, structural 373
equivalence 371
eqv 358, 371
eqvb 375
error handler 33
error tag 33
error value 33
error() 33
error 295
escape continuation 184, 294
eta expansion 389
eval 318
eval() 321
evaluation 228
  order of 210
evenp 376
existsp 258
existsp() 258
expand() 230, 236
expandbody() 232
expanddef() 234
expect() 34
expt 376

external representation 92, 115, 128
  canonical 93

—F—
fatal() 35
fetcharg() 45
filter 366
findsym() 81
fixed arguments 180, 288
fixnum 18, 68, 102, 267
fixnum arithmetic 239
fixnum object 103, 117
fixnum overflow 239
fixover() 239
fixp() 24
fixval() 68
flatargs() 134
flush() 38
fold 365
folding 365
foldr 365
foreach 368
formal argument list 133
formal argument 149
format 260
format() 260
fragmentation 61
frame pointer 172
free dots 111
free variable 138, 145, 149, 354
freevars() 145
function 115, 138, 148
  inline, binary 212
  inline, ternary 212
  inline, unary 211
  built-in 197
  lambda 160, 236
  higher order 353, 365
  immediate outer 140, 155
  local 354
  outer 140
  pure 156, 318
  primitive 94, 145, 196, 239, 296, 346
function application 136, 176, 203, 210
  tail application 178, 203
functional purity 318

—G—
garbage collection 20, 190
garbage collector 41, 47, 154, 192
gc 279
gc() 55
gcd 376
gcv() 63
general recursion 360
gensym 80, 221, 279
gensym() 279
global environment 33, 90
global program transformation 155
gottrace() 30
greatest common divisor 376
grtr() 243
gteq() 243

—H—
halt instruction 176
hash table 68, 72
hash() 74
head 19
header 271
heap 17
higher order function 353, 365
homoiconicity 128
host environment 168
ht 98
htadd() 78
htdata() 74
htelts() 74

htgrow() 76
htlen() 74
htlookup() 77
htrem() 78
htsize() 72
htslots() 74

—I—
identity 372
    symbol 79
if 132, 205
if* 132, 205, 356
image 270
image dumping 271
image file 345
image variable 273, 333
imghdr 271
immediate outer function 140, 155
immutability 99
implementation language 196
improper list 26
increment 363
indefinite extent 41, 139
init() 327
initialization map 141
initmap 141, 149, 206
initmap() 149
initrts() 324
inline operand 194
inlining 196, 239
inport object 120
inportp() 25
input/output 36
ins() 283
instruction 197, 296
    applis 180
    applist 180
    apply 177
    arg 175
    brf 176
    brt 176
    catch 183
    closure 182
    cparg 183
    cpref 183
    entcol 180
    enter 177
    halt 176
    jmp 176
    macro 184
    mkenv 183
    propenv 183
    push 176
    pushtrue 176
    pushval 176
    quote 44, 175
    ref 175
    return 177
    setarg 175
    setref 176
    tailapp 177
    throw 184
instruction code 27
instruction pointer 172
intdiv() 242
intern() 81
interning 80, 279
interpret() 319
interpretation 319
interpreter loop 295
interrupt 95
intrem() 242

—J—
jmp instruction 176
jump around jump 206

—K—
kbdintr() 323
key 72

—L—

labels 354
lambda 133
lambda function 160, 236
lambda lifting 154
lamconv() 153
lastpair() 150
lcm 376
lconc() 261
length 368
length() 130
less() 243
let 352, 360
    named 360
let* 359
letrec 355
lexical scoping 138
library 345
liftable() 158
liftargs() 159
liftlim 156
liftnames() 158
list 26, 99, 119, 128, 347
    dotted 26, 99, 119
    list 99
    improper 26
    of formal arguments 133
    proper 369
listp 369
listp() 26
liststr 264
liststr() 264
listvec 265
listvec() 265
literal
    character 14
    numeric 14
    string 14, 105
    vector 265
literal data object 186, 194
load 268
load() 269
loadfile() 268
loadimg() 276
loading 268
local binding 139, 279
local environment 139, 154, 172, 207
local functions 354
location 138, 176, 283, 319
lock_port() 88
logical AND 349
logical OR 356
longusage() 335
low-level macro transformer 227
lsubr0() 198
lsubr1() 198
lteq() 243

—M—

machine code, abstract 127
machine, virtual 168
macro 135, 234
macro applications 227
macro binding 184
macro body 184
macro expander 15
macro expansion 226, 401
macro instruction 184
main() 338
mapcar 353, 367
mapconv() 152
mapexp() 230
mapping 353
mark and sweep 43
mark() 52
marking 43
match 75
max 219, 391
memcmp_ci() 247

memory pools 18
memq 144, 367
memq() 133
meta command 108, 425
meta() 108
microcode 185, 296
min 219, 391
mkatom() 23
mkchar() 69
mkctag() 293
mkenv instruction 183
mkfix() 68
mkht() 72
mkport() 38, 71
mkstr 250
mkstr() 33, 69
mksym() 80
mkvec 254
mkvec() 70
mod 377
mul() 240
mulpg 241

—N—

named let 360
nandb 375
nconc 262
nconc() 85
negation 219
neutral element 365
newmacro() 230
newport() 86, 71
newvar() 151
newvars() 151
newvec() 65
nl 98
nl() 29
nlconc() 262
node collector 47
node pool 17
nodes 14, 18, 121
non-interactive compilation 410
non-local exits 291
non-local transfer 184
norb 375
normal form 228, 295, 318
notb 375
nreconc() 83
nrever 347
nreverse() 84
ntoa() 116
nullary inline functions 211
numeric literal 14
numstr 267
numstr() 267

—O—

obhash() 74
obindex() 191
object
  char 116
  closure 411
  fixnum 103, 117
  inport 120
  literal 186, 194
  outport 120
  port 71, 52, 120
  special 17, 103, 121
  string 118
  tagged 22
object table 15, 43, 187, 327, 345, 411
obslot() 190
octal char name 96
octalp() 95
octchar() 95
op1() 283
op2() 283
opcode 27, 197
open coding 197, 416

open-infile 258
open-outfile 258
open_inport() 86
open_outport() 86
openfile() 258
operand 284
operation codes 27
    inline 194
optional argument 180
or 356
orb 375
order of evaluation 210
osubr0() 198
osubr1() 198
outer function 140
outport object 120
outportp() 25
overflow, fixnum 239

—P—
pair 18, 99, 119, 191
pairp() 26
parser 128
    recursive descent 93
patch() 195
peekc 259
peephole optimization 419
performance 406
pointer reversal 51
pool size 273
pop() 285
port 36
port locking 88
port number 36, 86, 120
port object 71, 52, 120
portflags 36
portno() 24, 71
pos() 101
posq() 148
prchar() 116
prcomm() 122
predicate 24, 364
    comparison 219
    type 24
pretty printing 115, 125
prex() 123
prfix() 117
primitive form 226
primitive function 94, 145, 196, 239, 296, 346
prin 16, 93, 116, 260
prin() 125
prin() 125
princ 116, 260
princ() 125
print 378
print() 125
printb() 29
printer 115
printer limit 125
prints() 29, 40
prlist() 119
prog 135, 203
program fragment 172, 173
program register 171
propagation 142
propenv instruction 183
proper list 369
prot() 66
prport() 120
prquote() 122
prstr() 118
pruatom() 121
pruspec() 121
prvec() 120
pure function 156
push instruction 176
push() 285
pushtrue instruction 176
pushval instruction 176

—Q—

qquote 349
quasiquotation 160, 349
quasiquotation template 349
quotation 122
quote 136
quote instruction 44, 175
quoted() 121

—R—

range checking 297
rdchar() 98
rderror() 97
rdfix() 104
rdlist() 99
rdquote() 107
rdstr() 105
rdsymfix() 103
rdvec() 101
read 92, 115, 128, 259
read eval print loop 323
readc 259
readc() 37
reader 18, 79, 92, 127
readln 379
reconc() 83
recursion 176
  general 360
recursive descent parser 93
ref instruction 175
register 171, 282
rejectc() 38
rem 377
repl() 324
report() 31, 125
reset_stdports() 88
result 318
ret() 289
return instruction 177
rever 347
reverse lookup 80
reverse() 83
ring buffer 30
run() 295
runtime stack 15, 172, 282, 324, 333

—S—

save 386
saveimg() 272
scanfix() 102
scomp() 247
scomp_ci() 248
sconc 251
sconc() 251
scope
  dynamic 362
  lexical 138
scopy 378
semantics, operational 173
sequal 248
set_inport() 87
set_outport() 29, 87
set_union() 144
setarg instruction 175
setq 136, 156, 158, 204
setref instruction 176
setupenv() 206
sfill 254
sfill() 254
sgrtr 248
sgteq 248
shallow binding 90, 141
shlb 375
short circuit 349
shrb 375
siequal 249
sigrtr 249
sigteq 249
siless 249
silteq 249

simplicity 170
skip() 283, 296
slashification 117, 124
sless 248
slteq 248
sp 98
space character 111
special form, derived 226
special object 17, 103, 121
special variable 327
specialp() 17
spine 99, 288
splice 349
sref 252
sref() 252
sset 252
sset() 252
stack
    compile time 195, 206
    runtime 15, 172, 282, 324, 333
stack frame 172, 184, 291
stack machine 413
stack pointer 172
stackref() 283
stackset() 243, 283
start 332
start() 332
step 363
stkalloc() 284
storage 170
strcmp_ci() 97
string 19, 69, 247, 347
string literal 14, 105
string object 118
string() 24
stringlen() 24
stringp() 25
strlist 266
strlist() 266
strnum 102, 267
strnum() 267
structural equality 373
structural operational semantics 173
sub_ovfl() 68
subprog() 224
subr 145
subr0() 198
subr0() 211
subr1() 198
subr2() 198
subr3() 198
subroutines 197
subrp() 145, 202
substr 253
substr() 252
subvec 257
subvec() 257
surprise 318
swap() 196
symbol 19, 79, 103, 138, 263
    uninterned 221
symbol identity 79
symbol name 14
symbol table 15, 79, 327
symbolic() 96
symbolp() 25
symlen() 24
symlistp() 133
symname 264
symname() 24, 264
symref() 82
syncheck() 136
syntax analysis 129
syntax tree, abstract 43, 128

—T—

tag field 18
tag() 21
tagged data object 22
tail application 178, 203

tail application context 203
tail context 203
tail recursion 176
tailapp instruction 177
target architecture 168
template 349
terms() 335
ternary inline functions 212
terpri 378
throw 382
throw instruction 184
throw() 293
throwerr() 294
thunk 379
top level environment 142, 151, 170, 319
tortoise and hare 369
type checking 297
type predicate 24
type tag 22, 52

—U—
uint 17
unary inline functions 211
undefined 241
undefined value 115, 297
uninterned symbol 221
union 144
uniqlistp() 133
unlock_port() 89
unmark_vecs() 62
unprot() 66
unquote 349
unreadable 115
untag 121
untag() 280
untagging 121
unwind 382
unwind handler 382
unwind stack 381
upward continuation 294
usage() 335

—V—
value 228
    undefined 115, 297
variable 90, 138
    bound 138, 149
    free 138, 145, 149, 354
    image 273, 333
variadic 186
vconc 255
vconc() 255
veclen() 24
veclink() 24
veclist 266
veclist() 266
vecndx() 24
vecsize() 24, 62
vector 19, 20, 120, 191, 254, 347
vector cell 14
vector literals 265
vector objects 24, 59, 70
vector pool 17
vector pool compactor 61
vector size 60
vector() 24, 74
vectorp() 25
vfill 256
vfill() 256
virtual machine 168
vref 255
vset 256
vset() 256

—W—
white space character 246
whitec 246
whitespc() 246
with 361, 384

with-infile 379
with-inport 380
with-outfile 379
with-outport 380
writec 261
writec() 261
writec() 261

—X—
xfread() 275
xfwrite() 272
xorb 375
xprint() 124
xread() 113
xread2() 111
xsub 219
xsub() 240

—Y—

—Z—
zip() 231
zipenv() 166

—Å—

—Ä—

—Ö—